Proceedings of the 7th International Conference on PGAS Programming Models

Proceedings of the
7th International Conference on
PGAS Programming Models

3-4th October 2013, Edinburgh, UK

Edited by M Weiland, A Jackson & N Johnson

Published by The University of Edinburgh

ISBN: 978-0-9926615-0-2

Contents

Editor's Introduction

The 7th edition of the International Conference on Partitioned Global Address Space languages took place in Edinburgh on the 3rd and 4th October 2013. The conference brought together over 60 attendees from across the globe: leading researchers and scientists from North America, Europe and Japan addressed a wide range of topics relevant to PGAS languages and exascale computing.

These proceedings collate the papers that were accepted for publication. The first section of the proceedings is dedicated to the research papers, which represent substantial bodies of work and progress beyond the state-of-the art. The subsequent section contains the "hot" category: these are shorter papers that introduce work in progress. The proceedings conclude with the poster submissions.

PGAS2013 was a highly successful conference that presented significant progress in the area of PGAS research. We hope that the proceedings reflect this success and will bring the research to a wider audience.

Michèle Weiland, Adrian Jackson and Nick Johnson,
Editors

Research Papers

These papers were accepted into the research category and are substantive works demonstrating new results, ideas, applications and models.

A scalable deadlock detection algorithm for UPC collective operations

Indranil Roy, Glenn R. Luecke, James Coyle and Marina Kraeva

Iowa State University's High Performance Computing Group, Iowa State University, Ames, Iowa 50011, USA
iroy@iastate.edu, grl@iastate.edu, jjc@iastate.edu, kraeva@iastate.edu.

Abstract

Unified Parallel C (UPC) is a language used to write parallel programs for shared and distributed memory parallel computers. Deadlock detection in UPC programs requires detecting deadlocks that involve either locks, collective operations, or both. In this paper, a distributed deadlock detection algorithm for UPC programs that uses run-time analysis is presented. The algorithm detects deadlocks in collective operations using a distributed technique with O(1) run-time complexity. The correctness and optimality of the algorithm is proven. For completeness, the algorithm is extended to detect deadlocks involving both locks and collective operations by identifying insolvable dependency chains and cycles in a shared wait-for-graph (WFG). The algorithm is implemented in the run-time error detection tool UPC-CHECK and tested with over 150 functionality test cases. The scalability of this deadlock detection algorithm for UPC collective operations is experimentally verified using up to 8192 threads.

1 Introduction

Deadlocks in complex application programs are often difficult to locate and fix. Currently UPC-CHECK [3] and UPC-SPIN [4] are the only tools available for the detection of deadlocks in UPC programs. UPC-SPIN employs a model-checking method which inherently does not scale beyond a few threads. In addition, every time the program is modified, the model has to be updated. In contrast, UPC-CHECK uses the algorithm presented in this paper to automatically detect deadlocks at run-time for programs executing on thousands of threads.

This new algorithm not only detects deadlocks involving UPC collective operations, but also verifies the arguments passed to the collective operation for consistency. The run-time complexity of this algorithm is shown to be $O(1)$. The algorithm has been extended to detect deadlocks involving both collective operations and locks. The run-time complexity of the extended algorithm is $O(T)$, where T is the number of threads. Using this deadlock detection algorithm UPC-CHECK detects all deadlock error test cases from the UPC RTED test suite [2].

The rest of this paper is organized as follows. Section 2 provides the background of various existing deadlock detection techniques. In Section 3, a new algorithm to detect potential deadlocks due to incorrect usage of UPC collective operations is presented. The correctness and run-time complexity analysis of the algorithm are also provided. Section 4 describes the extended algorithm to detect deadlocks involving both locks and collective operations. The scalability of this deadlock detection algorithm is experimentally confirmed in Section 5. Finally, Section 6 contains the concluding remarks.

2 Background

Out-of-order calls to collective operations on different threads may create a deadlock. Even when the calls to collective operations are in-order, various non-local semantics dictate that consistent arguments need to be used in all participating threads. Non-adherence to these semantics could lead to a deadlock or departure from intended behavior of the program. However, building scalable tools to detect such errors remains a challenge.

Model-checking tools like MPI-SPIN [10] and UPC-SPIN [4] can detect all possible deadlock conditions arising from all combination of parameters in all possible control-flows. However, such tools cannot scale beyond a few threads due to the combinatorial state-space explosion. Tools employing dynamic formal verification methods do not check all the control flows and hence can be used for larger programs. Such tools ISP [13], MODIST [16] and POE [12] generally employ centralized deadlock detection schemes which limit them to verifying executions using a small number of processes. Execution time of such methods is also usually high. DAMPI [15] is a dynamic formal verification tool which overcomes this limitation by using a distributed heuristics-based deadlock detection algorithm.

The most practical method for detecting deadlocks in terms of scalability is run-time analysis. Tools using this kind of analysis detect only those deadlocks that would actually occur during the current execution of a program. Marmot [7] and MPI-CHECK [8] employ synchronized time-out based strategies to detect deadlock conditions. Time-out based strategies may report false-positive error cases and generally cannot pinpoint the exact reason for the error. On the other hand, the run-time analysis tool, Umpire [14] uses a centralized WFG based on the generalized $AND \oplus OR$ model developed by Hilbrich et al. [5]. However, MPI-CHECK, Marmot and Umpire are all based on the client-server model, which limits their scalability to a few hundred threads. In order to overcome this limitation, MUST [6] utilizes a flexible and efficient communication system to transfer records related to error detection between different processes or threads.

Our algorithm uses a different approach to detect deadlocks involving collective operations. We exploit two properties of operations in UPC which make deadlock detection easier than in MPI. Firstly, communication between two processes is non-blocking and secondly, non-determinism of point-to-point communication operations in terms of any_source cannot occur in UPC.

3 Detecting deadlocks due to collective errors in collective operations

Terms used throughout the rest of this paper are:

1. $THREADS$ is an integer variable that refers to the total number of threads with which the execution of the application was initiated.

2. A UPC *operation* is defined as any UPC statement or function listed in the UPC specification.

3. The *state* of a thread is defined as the name of the UPC operation that the thread has reached. In case the thread is executing an operation which is not a collective or lock-related UPC operation, the *state* is set to `unknown`. If the thread has completed execution, the *state* is set to `end_of_execution`.

4. A *single-valued* argument is an argument of a UPC collective operation which must be passed the same value on every thread.

5. The *signature* of a UPC operation on a thread consists of the name of the UPC operation and the values which are about to be passed to each of the single-valued arguments of the UPC collective operation on that thread.

6. For any thread k, s_k is a shared data structure which stores the state of thread k in field $s_k.op$. In case state is the name of a UPC collective operation, s_k also stores the single-valued arguments of the operation on that thread.

7. To *compare* the signatures of UPC operations stored in s_i and s_j means to check whether all the fields in s_i and s_j are identical.

8. If all the fields in s_i and s_j are identical, the result of the comparison is a *match*, otherwise there is a *mismatch*.

9. $C(n, k)$ denotes the n^{th} collective operation executed by thread k.

The UPC specification requires that the order of calls to UPC collective operations must be the same for all threads [11]. Additionally, each '*single-valued*' argument of a collective operation must have the same value on all threads. Therefore deadlocks involving only collective UPC operations can be created if:

1. different threads are waiting at different collective operations,

2. values passed to single-valued arguments of collective functions do not match across all threads, and

3. some threads are waiting at a collective operation while at least one thread has finished execution.

An algorithm to check whether any of the above 3 cases is going to occur must compare the collective operation which each thread is going to execute next and its single-valued arguments with those on other threads. Our algorithm achieves this by viewing the threads as if they were arranged in a circular ring. The left and right neighbors of a thread i are thread $(i-1)\%THREADS$ and thread $(i+1)\%THREADS$ respectively. Each thread checks whether its right neighbor has reached the same collective operation as itself. Since this checking goes around the whole ring, if all the threads arrive at the same collective operation, then each thread will be verified by its left neighbor and there will be no mismatches of the collective operations. However, if any thread comes to a collective operation which is not the same as that on the other thread, its left neighbor can identify the discrepancy, and issue an error message. The correctness of this approach is proven in Section 3.1.

On reaching a collective UPC operation, a thread k first records the signature of the collective operation in s_k. Thread k sets $s_k.op$ to `unknown` after exiting from a operation.Let a and b be the variables that store signatures of collective operations. The assign ($\leftarrow$) and the compare ($\ncong$) operations for the signatures of collective operation stored in a and b are defined as follows:

1. $b \leftarrow a$ means

   (a) assign value of variable $a.op$ to variable $b.op$, and

   (b) if $a.op \neq end\_of\_execution$, copy values of single-valued arguments recorded in a to b

2. $b \ncong a$ is true if

   (a) $b.op \neq a.op$, or

   (b) if $a.op \neq end\_of\_execution$, any of the single-valued arguments recorded in a is not identical to the corresponding argument recorded in b.

Let thread j be the right neighbor of thread i. During execution, thread i or thread j could reach their respective n^{th} collective operation first. If thread i reaches the operation first, then it cannot compare $C(n,i)$ recorded in s_i with $C(n,j)$, since s_j does not contain the signature of the n^{th} collective operation encountered on thread j, i.e. $C(n,j)$. The comparison can be delayed until thread j reaches its n^{th} collective operation. In order to implement this, another shared variable ds_k is used on each thread k to store the desired signature. For faster access, both shared variables s_k and ds_k have *affinity*[1] to thread k. If thread i finds that thread j has not reached a collective operation ($s_j.op$ is `unknown`), then it assigns s_i to ds_j. When thread j reaches a collective operation it first records the signature in s_j and then compares it with ds_j. If they do not match, then thread j issues an error message, otherwise it sets $ds_j.op$ to `unknown` and continues.

If thread i reaches the collective operation after thread j ($s_j.op$ is assigned a name of a collective UPC operation), then thread i compares s_j with s_i. If they match, then there is no error, so execution continues.

The UPC specification does not require collective operations to be synchronizing. This could result in one or more state variables on a thread being reassigned with the signature of the next collective operation that it encounters before the necessary checking is completed. To ensure that the signature of the n^{th} collective operation encountered on thread i i.e. $C(n,i)$ is compared with the signature of the n^{th} collective operation encountered on thread j, i.e. $C(n,j)$, the algorithm must ensure that:

1. If thread i reaches the n^{th} collective operation before thread j and assigns ds_j the signature of $C(n,i)$, it does not reassign ds_j before thread j has compared ds_j with s_j, and

2. If thread j reaches the n^{th} collective operation before thread i and assigns s_j the signature of $C(n,j)$, it does not reassign s_j before either thread i has a chance to compare it with s_i or thread j has a chance to compare it with ds_j.

In order to achieve the behavior described above, two shared variables r_s_j and r_ds_j are used for every thread j. Variable r_s_j is used to prevent thread j from reassigning s_j before the necessary comparisons described above are completed. Similarly, variable r_ds_j is used to prevent thread i from reassigning ds_j before the necessary comparisons are completed. Both r_s_j and r_ds_j have affinity to thread j.

For thread j, shared data structures s_j and ds_j are accessed by thread i and thread j. To avoid race conditions, accesses to s_j and ds_j are guarded using lock $L[j]$.

Our deadlock algorithm is implemented via the following three functions:

- *check_entry()* function which is called before each UPC operation to check whether executing the operation would cause a deadlock,

- *record_exit()* function which is called after each UPC operation to record that the operation is complete and record any additional information if required, and

[1] In UPC, shared variables that are stored in the physical memory of a thread are said to have *affinity* to that thread.

- *check_final()* function which is called before every `return` statement in the `main()` function and every `exit()` function to check for possible deadlock conditions due to the termination of this thread.

Algorithm A1

```
1  On thread i:
   // Initialization
2  s_i.op ← ds_i.op ← unknown, r_s_i ← 1, r_ds_j ← 1
   // Function definition of check_entry(f_sig):
3  if THREADS = 1 then
4      Exit check. ;
5  end
6  Acquire L[i] ;
7  s_i ← f_sig ;
8  r_s_i ← 0 ;
9  if ds_i.op ≠ unknown then
10     if ds_i ≇ s_i then
11         Print error and call global exit function. ;
12     end
13     r_s_i ← 1 ;
14     r_ds_i ← 1 ;
15     ds_i.op ← unknown ;
16 end
17 Release L[i] ;
18 Wait until r_ds_j = 1 ;
19 Acquire L[j] ;
20 if s_j.op = unknown then
21     ds_j ← s_i ;
22     r_ds_j ← 0 ;
23 else
24     if s_j ≇ s_i then
25         Print error and call global exit function ;
26     end
27     r_s_j ← 1 ;
28 end
29 Release L[j]
   // Function definition of check_exit():
30 Wait until r_s_i = 1 ;
31 Acquire L[i] ;
32 s_i.op ← unknown ;
33 Release L[i]
   // Function definition of check_final():
34 Acquire L[i] ;
35 if ds_i.op ≠ unknown then
36     Print error and call global exit function. ;
37 end
38 s_i.op ← end_of_execution ;
39 Release L[i] ;
```

The pseudo-code of the distributed algorithm[2] on each thread i to check deadlocks caused by incorrect or missing calls to collective operations[3] is presented.Function *check_entry*() receives as argument the signature of the collective operation that the thread has reached, namely f_sig.

3.1 Proof of Correctness

Using the same relation between thread i and thread j, i.e. thread i is the left neighbor of thread j, the proof of correctness is structured as follows. Firstly, it is proved that the algorithm is free of deadlocks and livelocks. Then Lemma 3.1 is used to prove that the left neighbor of any thread j does not reassign ds_j before thread j can compare s_j with ds_j. Lemma 3.2 proves that the right neighbor of any thread i, does not reassign s_j before thread i can compare s_i with s_j. Using Lemma 3.1 and Lemma 3.2 it is proven that for any two neighboring threads i and j, signature of $C(n, j)$ is compared to the signature of $C(n, i)$. Finally, using Lemma 3.3 the correctness of the algorithm is proven by showing that : 1) no error message is issued if all the threads have reached the same collective operation with the same signature and 2) an error message is issued if at least one thread has reached a collective operation with a signature

[2]As presented, the algorithm forces synchronization even for non-synchronizing UPC collective operations. However, if forced synchronization is a concern, this can be handled with a queue of states. This will not change the O(1) behavior.

[3]UPC-CHECK treats non-synchronizing collective operations as synchronizing operations because the UPC 1.2 specification says that "Some implementations may include unspecified synchronization between threads within collective operations" (footnote; page 9).

different from the signature of the collective operation on any other thread. Case 1 is proved by Theorem 3.4 and Case 2 is proved by Theorem 3.5.

There is no hold-and-wait condition in algorithm A1, hence there cannot be any deadlocks in the algorithm. To show that the algorithm is livelock-free, we show that any given thread must eventually exit the waits on line 18 and 30. For any thread i reaching its n^{th} collective operation $C(n,i)$, thread i can wait at line 18 if thread i itself had set r_ds_j to 0 on line 22 on reaching $C(n-1,i)$. This is possible only if thread i found that $s_j.op = unknown$ on line 20, i.e. thread j is not executing an UPC collective operation. Eventually thread j either reaches the end of execution or a UPC collective operation. In the former case, a deadlock condition is detected, an error message is issued and the application exits. In the second case, thread j finds conditional statement on line 9 to be true and sets r_ds_j to 1 on line 14. Since only thread i can set r_ds_j to 0 again, thread i would definitely exit the wait on line 18. Similarly, for thread j to be waiting at line 30 after executing $C(n,j)$, it must not have set r_s_j to 1 at line 13. This means that $ds_j.op$ must be equal to *unknown* at line 9, implying that thread i has still not executed line 21 and hence line 20 (by temporal ordering) due to the atomic nature of operations accorded by $L[j]$. When thread i finally acquires $L[j]$, the conditional statement on line 20 must evaluate to false. If thread i has reached a collective operation with a signature different from that of $C(n,j)$, a deadlock error message is issued, otherwise r_s_j is set to 1. Since only thread j can set r_s_j to 0 again, it must exit the waiting at line 30.

Lemma 3.1. *After thread i assigns the signature of $C(n,i)$ to ds_j, then thread i does not reassign ds_j before thread j compares s_j with ds_j.*

Proof. This situation arises only if thread i has reached a collective operation first. After thread i sets ds_j to s_i (which is already set to $C(n,i)$) at line 21, it sets r_ds_j to 0 at line 22. Thread i cannot reassign ds_j until r_ds_j is set to 1. Only thread j can set r_ds_j to 1 at line 14 after comparing s_j with ds_j. □

Lemma 3.2. *After thread j assigns the signature of $C(n,j)$ to s_j, then thread j does not reassign s_j before it is compared with s_i.*

Proof. After thread j assigns the signature of $C(n,j)$ to s_j at line 7, it sets r_s_j to 0. Thread j cannot modify s_j until r_s_j is set to 1. If thread i has already reached the collective operation, then thread j sets r_s_j to 1 at line 13 only after comparing s_j with ds_j at line 10. However, thread i must have copied the value of s_i to ds_j at line 21. Alternatively, thread j might have reached the collective operation first. In this case, thread i sets r_s_j to 1 at line 27 after comparing s_i to s_j at line 24. □

Lemma 3.3. *For any neighboring threads i and j, the signature of $C(n,i)$ is always compared with the signature of $C(n,j)$.*

Proof. This is proved using induction on the number of the collective operations encountered on threads i and j.

Basis. Consider the case where n equals 1, i.e. the first collective operation encountered on thread i and thread j. The signature of $C(1,i)$ is compared with the signature of $C(1,j)$. If thread i reaches collective operation $C(1,i)$ first, then it assigns ds_j the signature of $C(1,i)$. Using Lemma 3.1, thread i cannot reassign ds_j until ds_j is compared with s_j by thread j on reaching its first collective operation, $C(1,j)$. Alternatively, if thread j reaches its collective operation first, then Lemma 3.2 states that after thread j assigns the signature of $C(1,j)$ to s_j, thread j cannot reassign s_j before it is compared with s_i. The comparison between s_j and s_i is

done by thread i after it reaches its first collective operation and has assigned s_i the signature of $C(1, i)$.

Inductive step. If the signature of $C(n, i)$ is compared with the signature of $C(n, j)$, then it can be proven that the signature of $C(n+1, i)$ is compared with the signature of $C(n+1, j)$. If thread i reaches its next collective operation $C(n+1, i)$ first, then it assigns ds_j the signature of $C(n+1, i)$. Using Lemma 3.1, thread i cannot reassign ds_j until ds_j is compared with s_j by thread j on reaching its next collective operation, i.e. $C(n+1, j)$. Alternatively, if thread j reaches its next collective operation first, then Lemma 3.2 states that after thread j assigns $C(n+1, j)$ to s_j, thread j cannot reassign s_j before it is compared with s_i. The comparison of s_j with s_i is done by thread i after it reaches its next collective operation and has asigned s_i the signature of $C(n+1, i)$. □

Using Lemma 3.3, it is proven that for any neighboring thread pair i and j, the signature of n^{th} collective operation of thread i is compared with the signature of n^{th} collective operation of thread j. As j varies from 0 to $THREADS - 1$, it can be said that when the n^{th} collective operation is encountered on any thread, it is checked against the n^{th} encountered collective operation on every other thread before proceeding. Thus in the following proofs, we need to only concentrate on a single (potentially different) collective operation on each thread. In the following proofs, let the signature of the collective operation encountered on a thread k be denoted by S_k. If a state or desired state $a_i.op$ is `unknown`, then it is denoted as $a = U$ for succinctness. Then in algorithm $A1$, after assigning the signature of the encountered collective operation, i.e. line $s_i \leftarrow f\_sig$, notice that for thread i:
s_i must be S_i,
ds_i must be either U or S_{i-1},
s_j must be either U or S_j, and
ds_j must be U.

Theorem 3.4. *If all the threads arrive at the same collective operation, and the collective operation has the same signature on all threads, then Algorithm A1 will not issue an error message.*

Proof. If $THREADS$ is 1, no error message is issued, so we need to consider only cases of execution when $THREADS > 1$. If all threads arrive at the same collective operation with the same signature, then during the checks after $s_i \leftarrow f\_sig$, is the same for all i. Let S denote this common signature. We will prove this theorem by contradiction. An error message is printed only if:

1. $ds_i \neq U$ and $ds_i \neq s_i \Rightarrow ds_i = S$ and $ds_i \neq S \Rightarrow S \neq S$ (contradiction) or
2. $s_j \neq U$ and $s_j \neq s_i \Rightarrow s_j = S$ and $s_j \neq S \Rightarrow S \neq S$ (contradiction)

So Theorem 3.4 is proved. □

Theorem 3.5. *If any thread has reached a collective operation with a signature different from the signature of the collective operation on any other thread, then a deadlock error message is issued.*

Proof. There can be a mismatch in the collective operation or its signature only if there is more than one thread.

Since the signatures of the collective operations reached on every thread are not identical, there must be some thread i for which $S_i \ncong S_j$. For these threads i and j, the following procedures are made to be atomic and mutually exclusive through use of lock $L[j]$:

- Action 1: Thread i checks s_j. If $s_j = U$, then thread i executes $ds_j \leftarrow s_i$, else, computes $s_j \ncong s_i$ and issues an error message if true.
- Action 2: Thread j assigns the signature of the collective operation it has reached to s_j. Thread j checks ds_j. If $ds_j \neq U$, the thread j computes $ds_j \ncong s_j$ and issues message if true.

There are only two possible cases of execution: either action 1 is followed by action 2 or vice versa.

In the first case, in action 1, thread i finds $s_j = U$ is true, executes $ds_j \leftarrow S_i$ and continues. Then in action 2, thread j executes $s_j \leftarrow S_j$, finds that $ds_j \neq U$ and hence computes $ds_j \ncong s_j$. Now, since $ds_j = S_i$ and $s_j = S_j$ and $S_i \neq S_j$ (by assumption) implies that $ds_j \ncong s_j$ is true. Therefore thread j issues an error message.

In the second case, in action 2, thread j assigns $s_j \leftarrow S_j$, finds $ds_j = U$ and continues. Before thread i initiates action 1 by acquiring $L[j]$, it must have executed $s_i \leftarrow S_i$. If $ds_i \neq U$ and $ds_i \ncong s_i$, then an error message is issued by thread i, otherwise it initiates action 1. Thread i finds $s_j \neq U$ and computes $s_j \ncong s_i$. Now, since $s_i = S_i$ and $s_j = S_j$ and $S_i \ncong S_j$ (by assumption) implies that $s_j \ncong s_j$ is true. Therefore thread i issues an error message.

Since the above two cases are exhaustive, an error is always issued if $S_i \ncong S_j$ and hence Theorem 3.5 is proved.

□

Theorem 3.6. *The complexity of the Algorithm A*1 *is O(1).*

Proof. There are two parts to this proof.

1. The execution-time overhead for any thread i is O(1). Any thread i computes a fixed number of instructions before entering and after exiting a collective operation. It waits for at most two locks $L[i]$ and $L[j]$ each of which can have a dependency chain containing only one thread, namely thread $i-1$ and thread j respectively. Thread i synchronizes with only two threads, i.e. its left neighbor thread $i-1$ and right neighbor thread j. There is no access to variables or locks from any other thread. Therefore the execution time complexity of the algorithm in terms of the number of threads is O(1).
2. The memory overhead of any thread i is independent of the number of threads and is constant.

□

3.2 Detecting deadlock errors involving upc_notify and upc_wait operations

The compound statement {*upc_notify; upc_wait*} forms a split barrier in UPC. The UPC specification requires that firstly, there should be a strictly alternating sequence of upc_notify and upc_wait calls, starting with a upc_notify call and ending with a upc_wait call. Secondly, there can be no collective operation between a upc_notify and its corresponding upc_wait call. These conditions are checked using a private binary flag on each thread which is set when a upc_notify statement is encountered and reset when a upc_wait statement is encountered. This binary flag is initially reset. If any collective operation other than upc_wait is encountered when the flag is set, then there must be an error. Similarly, if a upc_wait statement is encountered when the flag is reset, then there must be an error. Finally, if the execution ends, while the flag

is set, then there must be an error. These checks are performed along with the above algorithm and do not require any communication between threads. Also modifying and checking private flags is an operation with complexity of O(1).

If all the threads issue the upc_notify statement, then the next UPC collective operation issued on all the threads must be a upc_wait statement. Therefore algorithm $A1$ working in unison with the above check needs to only verify the correct ordering of upc_notify across all threads. The correct ordering of the upc_wait statements across all threads is automatically guaranteed with the above mentioned checks. This is reflected in Algorithm $A2$.

4 Detecting deadlocks created by hold-and-wait dependency chains for acquiring locks

In UPC, acquiring a lock with a call to the *upc_lock*() function is a blocking operation. In UPC program, deadlocks involving locks occur when there exists one of the following conditions:

1. a cycle of hold-and-wait dependencies with at least two threads, or
2. a chain of hold-and-wait dependencies ending in a lock held by a thread which has completed execution, or
3. a chain of hold-and-wait dependencies ending in a lock held by a thread which is blocked at a synchronizing collective UPC operation.

Our algorithm uses a simple edge-chasing method to detect deadlocks involving locks in UPC programs. Before a thread u tries to acquire a lock, it checks if the lock is free or not. If it is free, the thread continues execution. Otherwise, if the lock is held by thread v, thread u checks $s_v.op$ to check if thread v:

1. is not executing a collective UPC operation or `upc_lock` operation ($s_v.op$ is *unknown*), or
2. is waiting to acquire a lock, or
3. has completed execution, or
4. is waiting at a synchronizing collective UPC operation.

If thread v is waiting to acquire a lock, then thread u continues to check the *state* of the next thread in the chain of dependencies. If thread u finally reaches thread m which is not executing a collective UPC operation or `upc_lock` operation, then no deadlock is detected. If thread u finds itself along the chain dependencies, then it reports a deadlock condition. Similarly, if thread u finds thread w which has completed execution at the end of the chain of dependencies, then it issues an error message.

When the chain of dependencies ends with a thread waiting at a collective synchronizing operation, the deadlock detection algorithm needs to identify whether the thread will finish executing the collective operation or not. Figure 1 illustrates these two cases. Thread u is trying to acquire a lock in a chain of dependencies ending with thread w. When thread u checks the $s_w.op$ of thread w, thread w may (a) not have returned from the n^{th} synchronizing collective operation $C_s(n, w)$, (b) have returned from the n^{th} synchronizing collective operation but has not updated the $s_w.op$ in the *check_exit*() function, (c) have completed executing *check_entry*() function for the next synchronizing collective operation $C_s(n+1, w)$, or (d) waiting at the $(n+1)^{th}$ synchronizing collective operation $C_s(n+1, w)$. The n^{th} synchronizing collective

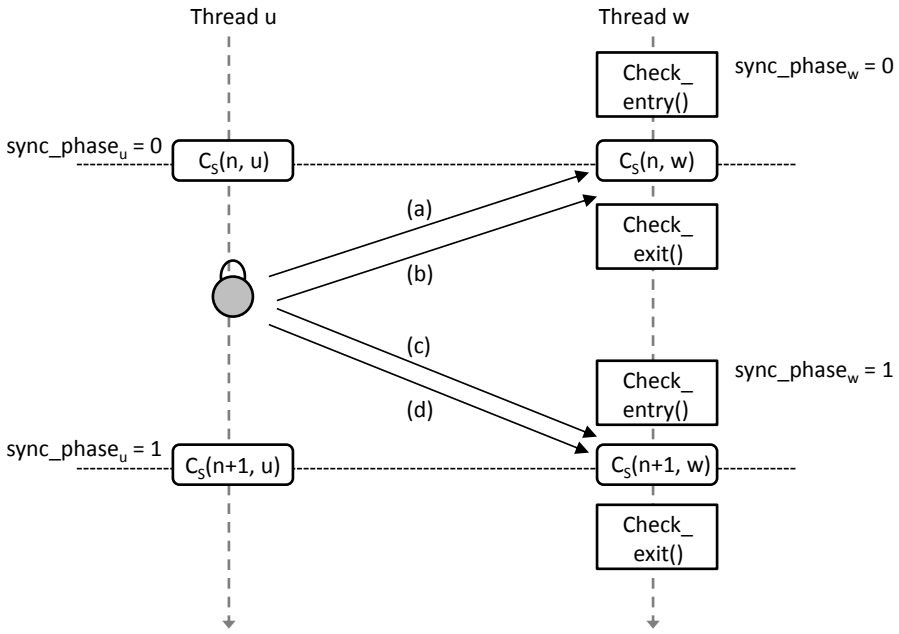


Figure 1: Possible scenarios when detecting deadlocks involving chain of hold-and wait dependencies. Scenario (a) or (b) is not a deadlock condition, while scenario (c) or (d) is.

operation encountered on thread w must be a valid synchronization operation that all threads must have called (otherwise the *check_entry*() function would have issued an error message). Therefore scenarios (a) and (b) are not deadlock conditions, while (c) and (d) are. To identify and differentiate between these scenarios, a binary shared variable $sync\_phase_k$ is introduced for each thread k. Initially $sync\_phase_k$ is set to 0 for all threads. At the beginning of each *check_entry*() function on thread k, the value $sync\_phase_k$ is toggled. Thread u can now identify the scenarios by just comparing $sync\_phase_u$ and $sync\_phase_w$. If they match (are *in-phase*), then it is either scenario (a) or (b) and hence no deadlock error message is issued. If they do not match (are *out-of-phase*), then it is either scenario (c) or (d) and hence a deadlock error message is issued.

4.1 The complete deadlock detection algorithm

The complete algorithm to detect deadlocks created by errors in collective operations and hold-and-wait dependency chains for acquiring locks is presented below. The *check_entry*() and *check_exit*() functions receive two arguments: 1) the signature of the UPC operation that the thread has reached, namely *f_sig* and 2) the pointer *L_ptr*. *L_ptr* points to the lock which the thread is trying to acquire or release if the thread has reached a *upc_lock*, *upc_lock_attempt* or *upc_unlock* statement.

Checking for dependency chains and cycles adds only a constant amount of time overhead for each thread in the chain or cycle. This means that the overhead is O(T) where T is the number of threads in the dependency chain.

Algorithm A2

```
1  On thread i:
   // Initialization
2  Create empty list of acquired and requested locks ;
3  s_i.op ← ds_i.op ← unknown; r_s_i ← 1; r_ds_j ← 1;
   (sync_phase_i ← 0)
   // Function definition of check_entry(f_sig, L_ptr):
4  Acquire L[i] ;
5  s_i ← f_sig ;
6  Release L[i] ;
7  if f_sig.op = at_upc_wait_statement then
8      Exit check ;
9  end
10 if f_sig.op = at_upc_lock_operation then
11     Acquire c_L ;
12     Check status of L_ptr ;
13     if L_ptr is held by this thread or is part of a cycle
       or chain of dependencies then
14         Print suitable error and call global exit ;
15     else
16         Update list of requested locks ;
17         Release c_L ;
18         Exit check ;
19     end
20 end
21 if f_sig.op = at_upc_unlock_operation then
22     if L_ptr is not held by this thread then
23         Print suitable error and call global exit ;
24     else
25         Update list of acquired locks ;
26         Exit check
27     end
28 end
   // Thread has reached a collective operation
29 if THREADS = 1 then
30     Exit check ;
31 end
32 Acquire c_L ;
33 if this thread holds locks which are in the list of
   requested locks then
34     Print suitable error and call global exit ;
35 end
36 Release c_L ;
37 Acquire L[i] ;
38 r_s_i ← 0 ;
39 if this is a synchronizing collective operation then
40     sync_phase_i ← (sync_phase_i + 1)%2 ;
41 end
42 if ds_i.op ≠ unknown then
43     if ds_i ≇ s_i then
44         Print error and call global exit function ;
45     end
46     r_s_i ← 1 ;
47     r_ds_i ← 1 ;
48     ds_i.op ← unknown ;
49 end
50 Wait until r_ds_j = 1 ;
51 Acquire lock L[j] ;
52 if s_j.op = unknown then
53     ds_j ← s_i ;
54     r_ds_j ← 0 ;
55 else
56     if s_j ≇ s_i then
57         Print error and call global exit function ;
58     end
59     r_s_j ← 1 ;
60 end
61 Release lock L[j]
   // Function definition of check_exit(f_sig, L_ptr):
62 Wait until r_s_i = 1 ;
63 Acquire L[i] ;
64 s_i ← unknown ;
65 Release L[i] ;
66 if f_sig.op = at_upc_lock_operation then
67     Acquire c_L ;
68     Remove L_ptr from the list of requested locks ;
69     Add L_ptr to the list of acquired locks ;
70     Release c_L ;
71 end
72 if f_sig.op = at_upc_lock_attempt_operation then
73     if L_ptr was achieved then
74         Acquire c_L ;
75         Remove L_ptr from the list of requested locks ;
76         Add L_ptr to the list of acquired locks ;
77         Release c_L ;
78     end
79 end
80 Continue execution ;
   // Function definition of check_final():
81 Acquire L[i] ;
82 s_i ← end_of_execution ;
83 if ds_i.op ≠ unknown then
84     Print error and call global exit function ;
85 end
86 Release L[i] ;
87 Acquire c_L ;
88 if this thread holds locks which are in the list of
   requested locks then
89     Print suitable error and call global exit ;
90 end
91 if this thread is still holding locks then
92     Print suitable warning ;
93 end
94 Release c_L ;
```

5 Experimental verification of scalability

This deadlock detection algorithm has been implemented in the UPC-CHECK tool [3]. UPC-CHECK was used to experimentally verify the scalability of this algorithm on a Cray XE6 machine running the CLE 4.1 operating system. Each node has two 16-core Interlagos processors. Since we are interested in the verification of scalability, the authors measured the overhead of our deadlock detection method for 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 and 8192 threads. The verification of scalability was carried out by first measuring the overhead incurred when calling a UPC collective operation and then measuring the overhead when running the CG and IS UPC NAS Parallel Benchmarks (NPB) [1]. The Cray C 8.0.4 compiler was used with the `-hupc` option. To pin processes and memory the aprun command was used with the following options:

`-ss -cc cpu`.

The authors first measured the overhead of checking for deadlocks involving the `upc_all_broadcast` operation with a message consisting of one 4 byte integer. Since deadlock checking is independent of the message size, the small message size was used so that the checking overhead could be easily measured. To measure the time accurately, 10,000 calls to `upc_all_broadcast` were timed and an average reported.

```
time (t1);
for (i = 0; i < 10000; i++)
{
    upc_all_broadcast;
}
time {t2};
bcast_time = (t2 - t1)/10000;
```

Overhead times ranged from 76 to 123 microseconds for multiple nodes, i.e. 64, 128, 256, 512, 1024, 2048, 4096 and 8192 threads. When replacing `upc_all_broadcast` with `upc_all_gather_all`, overhead times ranged from 73 to 119 microseconds. In both cases, a slight increase is observed as we increase the number of threads. The authors attribute this to the fact that, in general, not all pairs of UPC threads can be mapped to physical processors for which the communication between UPC threads i and $(i+1)\%THREADS$ is the same for all i. The maximal communication time for optimally placed UPC threads still grows slowly as the total number of UPC threads grows. The deviation from constant time in the above experiment is only a factor of 1.5 for 128 times as many UPC threads.

Number of threads	Class B			Class C		
	Without checks	**With checks**	**Overhead**	**Without checks**	**With checks**	**Overhead**
2	77.2	77.6	0.4	211.2	211.8	0.6
4	41.4	41.7	0.3	112.7	112.8	0.1
8	28.1	28.7	0.6	73.9	74.2	0.3
16	15.3	16.0	0.6	39.4	40.0	0.6
32	8.6	9.5	0.9	21.1	22.1	0.9
64	5.5	6.6	1.1	13.1	14.0	1.0
128	3.3	4.7	1.3	8.3	9.7	1.4
256	NA	NA	NA	5.6	7.2	1.6

Table 1: Time in seconds of the UPC NPB-CG benchmark with and without deadlock checking

Number of threads	Class B			Class C		
	Without checks	With checks	Overhead	Without checks	With checks	Overhead
2	4.56	4.59	0.03	20.00	20.11	0.11
4	2.18	2.18	0.00	9.50	9.52	0.01
8	1.34	1.34	0.00	5.28	5.28	0.00
16	0.79	0.79	0.00	3.46	3.46	0.00
32	0.42	0.43	0.01	1.89	1.89	0.00
64	0.29	0.30	0.01	1.30	1.31	0.01
128	0.21	0.22	0.01	0.82	0.82	0.00
256	0.26	0.27	0.01	0.57	0.57	0.00

Table 2: Time in seconds of the UPC NPB-IS benchmark with and without deadlock checking

UPC-CHECK was tested for correctness using 150 tests from the UPC RTED test suite [2]. Each test contains a single deadlock. For all the tests, UPC-CHECK detects the error, prevents the deadlock from happening and exits after reporting the error correctly [3]. Since these tests are very small, the observed overhead was so small that we could not measure them accurately.

Timing results for the UPC NPB CG and IS benchmarks are presented in Tables 1 and 2 using 2, 4, 8, 16, 32, 64, 128, and 256 threads. Timings using more than 256 threads could not be obtained since these benchmarks are written in a way that prevents them from being run with more than 256 threads. These results also demonstrate the scalability of the deadlock detection algorithm presented in this paper. Timing data for the class B CG benchmark using 256 threads could not be obtained since the problem size is too small to be run with 256 threads.

6 Conclusion

In this paper, a new distributed and scalable deadlock detection algorithm for UPC collective operations is presented. The algorithm has been proven to be correct and to have a run-time complexity of O(1). This algorithm has been extended to detect deadlocks involving locks with a run-time complexity of O(T), T is the number of threads involved in the deadlock. The algorithm has been implemented in the run-time error detection tool UPC-CHECK and tested with over 150 functionality test cases. The scalability of this deadlock detection algorithm has been experimentally verified using up to 8192 threads.

In UPC-CHECK, the algorithm is implemented through automatic instrumentation of the application via a source-to-source translator created using the ROSE toolkit [9]. Alternatively, such error detection capability may be added during the precompilation step of a UPC compiler. This capability could be enabled using a compiler option and may be used during the entire debugging process as the observed memory and execution time overhead even for a large number of threads is quite low.

Acknowledgment

This work was supported by the United States Department of Defense & used resources of the Extreme Scale Systems Center at Oak Ridge National Laboratory.

References

[1] UPC NAS Parallel Benchmarks.

[2] James Coyle, James Hoekstra, Marina Kraeva, Glenn R. Luecke, Elizabeth Kleiman, Varun Srinivas, Alok Tripathi, Olga Weiss, Andre Wehe, Ying Xu, and Melissa Yahya. UPC run-time error detection test suite. 2008.

[3] James Coyle, Indranil Roy, Marina Kraeva, and Glenn Luecke. UPC-CHECK: a scalable tool for detecting run-time errors in Unified Parallel C. *Computer Science - Research and Development*, pages 1–7. 10.1007/s00450-012-0214-4.

[4] Ali Ebnenasir. UPC-SPIN: A Framework for the Model Checking of UPC Programs. In *Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS '11, 2011.

[5] Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. A graph based approach for MPI deadlock detection. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 296–305, New York, NY, USA, 2009. ACM.

[6] Tobias Hilbrich, Martin Schulz, Bronis R. Supinski, and Matthias S. Müller. MUST: A scalable approach to runtime error detection in mpi programs. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 53–66. Springer Berlin Heidelberg, 2010. 10.1007/978-3-642-11261-4_5.

[7] Bettina Krammer, Matthias Müller, and Michael Resch. MPI application development using the analysis tool MARMOT. In Marian Bubak, Geert van Albada, Peter Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, volume 3038 of *Lecture Notes in Computer Science*, pages 464–471. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24688-6_61.

[8] Glenn R. Luecke, Yan Zou, James Coyle, Jim Hoekstra, and Marina Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14(11):911–932, 2002.

[9] Daniel J. Quinlan and et al. ROSE compiler project.

[10] Stephen Siegel. Verifying parallel programs with MPI-Spin. In Franck Cappello, Thomas Herault, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 13–14. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-75416-9_8.

[11] The UPC Consortium. UPC Language Specifications (v1.2). 2005.

[12] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 66–79, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: a tool for model checking mpi programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 285–286, New York, NY, USA, 2008. ACM.

[14] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[15] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B.R. de Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1 –10, nov. 2010.

[16] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

Fortran coarray library for 3D cellular automata microstructure simulation

Anton Shterenlikht

Mech Eng Dept, University of Bristol, University Walk, Bristol BS8 1TR, UK
mexas@bris.ac.uk

Abstract

Coarrays are a part of Fortran language standard. In this work coarrays are used to construct a parallel cellular automata Fortran library for microstructure simulation. The library uses a range of multi-dimensional real, integer and logical coarrays with a three-dimensional image grid. A range of coarray synchronisation and collective statements are used. The performance of the library is measured using several test programs on HECToR, including solidification and fracture simulations. Models in excess of 10^{10} cells have been successfully analysed on up to $2^{15} = 32768$ cores with a speed-up of 20 compared to runs on 512 cores. While the library delivers useful microstructural predictions, the scaling results are disappointing, indicating the need for code optimisation. The library is freely available, under BSD license, from: http://eis.bris.ac.uk/∼mexas/cgpack.

1 Cellular Automata for Microstructure Simulation

Cellular automata (CA) is a popular microstructure simulation tool. It is a discrete space – discrete time method, in which space is divided into many small identical cells with a number of pre-defined states. State of each cell changes based on the states of some neighbouring cells. A CA model is often coupled with a finite element model to result in a hybrid discrete/continuum, multi-scale mechanical model. Many such hybrid models have been successfully used for the prediction of the ductile to brittle fracture [9], oxide cracking in hot rolling [3], grain instability [7], solidification [4] and recrystallisation [8]. Previously we have created a serial CA micro- and nano-structure evolution library, implemented in Fortran 2003 [7]. In this paper we present an attempt to parallelise this library with Fortran 2008 coarrays.

2 Fortran Coarrays

Coarrays are a part of the Fortran 2008 standard [5]. Coarray collectives are at a draft technical specification stage [6] and are expected to be part of the next minor revision of the standard. However, Cray Fortran compiler provides coarray collectives as an extension to the standard, with syntax and function very close to the technical specification. At this time coarrays are fully supported only by Cray and Intel compilers. G95 and GCC compilers provide partial coarray support (e.g. GCC supports only a single image) [1].

Coarrays are intended for SIMD type parallel programs. Multiple copies of the executable (*images*) are distributed to processors. If an array is declared as a coarray then a copy of this array is created on every processor. Each processor has full read/write access to coarray elements on all processors. For example, `real :: a(2,3)[*], b[*]` declares a 2×3 real *array coarray* and a real *scalar coarray*. Any processor can then read/write from/to a coarray on any image, including itself, using coarray *cosubscript*. A reference to a variable with no cosubscript is always a reference to the local variable. For example, line `b=a(1,1)[3]`, executed by any

image, will copy element `(1,1)` of array `a` from image 3 to local variable `b`; line `a(2,:)=b[1]`, executed by any image, will set all elements in row 2 of array `a` on that image to the value of `b` on image 1. The total number of images is typically set at run-time via environment variables and is available to the programmer via `num_images` intrinsic. The programmer can control which image executes what lines using `this_image` intrinsic.

While all local arrays declared as coarrays are independent, together they can be thought to represent one large global array. The interpretation of such imaginary global array is entirely up to the user. Thus in the coarray model, the global address space is not partitioned but rather *assembled*.

3 CA Coarray Data Structures

The major CA model data structure is the cellular array itself. We define it as a 4D coarray with a 3D image grid: `integer, allocatable :: space(:,:,:,:)[:,:,:]`. The 3D image grid is chosen because it minimises the amount of internal halo exchange information. The array has three spatial dimensions plus an extra dimension to store multiple physical quantities (layers). At present the library provides for two layers: (1) grains and (2) damage.

In addition to the main `space` coarray, all other data structures which need to be accessed by all images are defined as coarrays. Examples are: grain volume coarray: `integer, allocatable :: gv(:)[:,:,:]` and coarray of grain rotation tensors: `real, allocatable :: rt(:,:,:)[:,:,:]`. Note that it makes sense to allocate these coarrays with the same *codimensions*.

4 Halo Exchange

Each cell in the model represents a cube of material at some user-defined scale. We use a 26-cell nearest neighbourhood, i.e. $3 \times 3 \times 3$ cells minus the central cell. Part of the neighbourhood of a boundary cell resides in an array on another image, hence halo exchange between neighbouring images is necessary before every cell evolution (solidification, coarsening, fracture, etc.) increment. To store the halos, the `space` coarray is allocated with size increased by 2 cells in each direction. The halo exchange is done in parallel. Consider the following simple halo exchange code, running on `n` images, where the user interprets the `space` coarray as sharing a common face normal to direction 1.

```
allocate( space(0:11,0:11,0:11) [n,1,*] )
integer :: imgpos(3), lcob(3), ucob(3)
imgpos = this_image( space )             ! Image location in the coarray grid
lcob = lcobound( space )                 ! Lower cobound of space coarray
ucob = ucobound( space )                 ! Upper cobound of space coarray
if ( imgpos(1) .ne. lcob(1) ) &          ! For all images but the leftmost
 space(0, 1:10,1:10) =        &          ! copy the upper boundary cell states
 space(10,1:10,1:10)          &          ! into the lower halo array
  [imgpos(1)-1,imgpos(2),imgpos(3)]
if ( imgpos(1) .ne. ucob(1) ) &          ! For all images but the rightmost
 space(11,1:10,1:10) =        &          ! copy the lower boundary cell states
 space(1, 1:10,1:10)          &          ! into the upper halo array
  [imgpos(1)+1,imgpos(2),imgpos(3)]
```

In this example each image has 10^3 microstructure cells plus 6(faces) $\times$ 100 + 12(edges) $\times$ 10 + 8(corners) = 730 halo cells. The production code (`hxi`) is a lot longer because of the need to exchange halos is three directions plus edges and corners. The library also includes a global halo exchange routine, (`hxg`) called when self-similar boundary conditions on the whole model are chosen by the user.

5 I/O

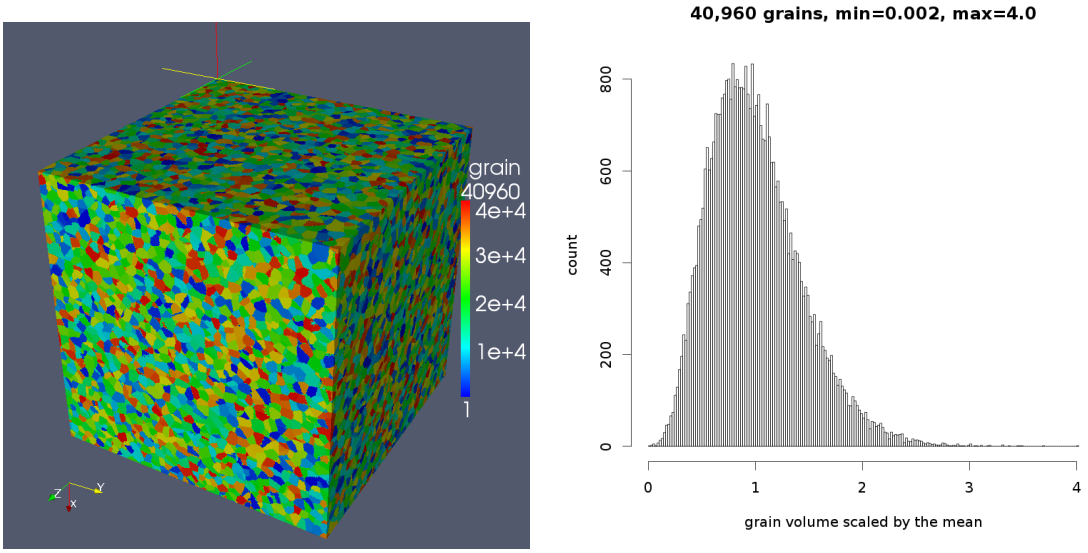
Coarray I/O can be done in at least two ways. Each image can write its own coarray array in a unique file. Then it is a job of the post-processing program to read and process all these files in the correct order. This approach is particularly attractive if the post-processing program supports parallel I/O. For example, Paraview (http://paraview.org) provides a parallel XDMF (http://xdmf.org) reader. This option has not be explored yet. Instead, we have implemented a serial routine that writes data from all images to a single binary file in correct order, as shown below. We then use the Paraview binary reader for visualisation. This is not very efficient, but simple. As long as the state of the model is written to file occasionally, this option seems acceptable. However, if the model has to be written to file often, then this serial routine will become a bottleneck.

```
lb = lbound(coarray) + 1                          ! Lower and upper bounds of
ub = ubound(coarray) - 1                          ! the coarray with no halos.
lcob = lcobound(coarray)                          ! Lower and upper cobounds
ucob = ucobound(coarray)                          !

if (this_image() .eq. 1) then                     ! Only img 1 does the writing

open(unit=iounit, file=fname, access="stream", & ! Open file for binary
 form = "unformatted", status = "replace")        ! stream write access

do coi3 = lcob(3), ucob(3)                        ! Nested loops
 do i3 = lb(3), ub(3)                             ! for writing
  do coi2 = lcob(2), ucob(2)                      ! in correct order
   do i2 = lb(2), ub(2)                           ! from all images.
    do coi1 = lcob(1), ucob(1)                    !
     write( unit = iounit )        &              ! Write one column at a time
      coarray(lb(1):ub(1),i2,i3)
       [coi1, coi2, coi3]
    end do
   end do
  end do
 end do
end do

end if
```

6 Model predictions

The library is distributed with a number of test programs, each calling different combinations of the library routines. The tests were done on HECToR phase 3, which is a Cray XE6 computer (www.hector.ac.uk). All tests were run with high resolution, 10^5 cells per grain, required to achieve scale independence [7].

Fig. 1(a) shows the predicted *equiaxed* microstructure typically found in normalised steels. This was obtained with `space(200,200,200)[8,8,8]` i.e. using 4.1×10^9 cells and 40,960 grains. On 512 processors (16 nodes, 32 processors per node) the run took 5m 36s wall time. The resulting output file was 16GB. Fig. 1(b) shows grain size (volume) histogram obtained from the solidification model shown in Fig. 1(a). This data is vital for model validation because it can be directly compared against experiments. Fig. 2(a) shows the histogram of the grain mis-orientation angle, for a material with no texture. The mis-orientation angle is an important parameter affecting grain boundary migration and fracture behaviour. The histogram agrees perfectly with the theoretical distribution. Finally Fig. 2(b) shows a cleavage crack propagating through a single, randomly oriented grain. The direction and the value of the maximum principal stress comes from the finite element solver. The importance of this result is in demonstrating the fracture predictive capability of the CA fracture model.

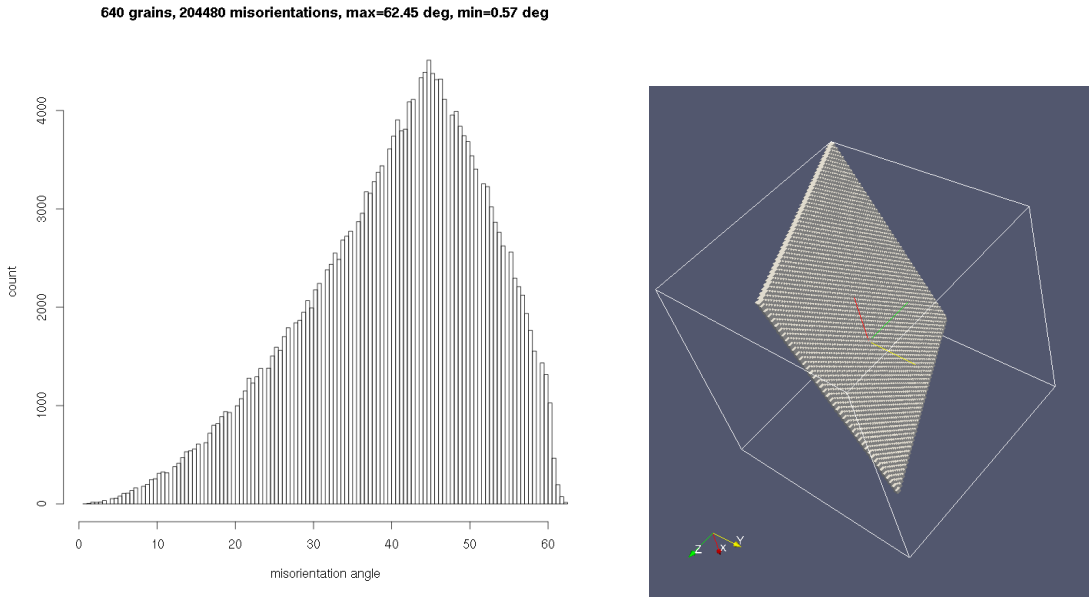


(a) (b)

Figure 1: A 4.1×10^9 cell, 40,960 grain equiaxed microstructure model, showing (a) grain arrangement with colour denoting orientation; (b) grain size size (volume) histogram.

7 Synchronisation, Performance and Code Optimisation

The Fortran standard provides a global barrier, `sync all`, and selective synchronisation between arbitrary image sets via `sync images`. Both types are used in this CA library. In addition, allocating or deallocating a coarray causes implicit synchronisation of all images.



(a) (b)

Figure 2: Examples of the CA coarray library use for microstructural modelling, showing (a) a grain mis-orientation histogram and (b) a cleavage crack in a randomly oriented single grain.

Synchronising a library is hard. This is because the order in which different routines can be called, if called at all, is not predictable. In addition, the user might write code where different images call different routines, potentially causing deadlocks.

One extreme option is to synchronise every routine on entry and exit, which, of course, is a performance killer, but removes the responsibility from the user to ever worry about synchronisation, provided they call the same routines in the same order from all images. Another extreme is not to provide any synchronisation in the library routines, but delegate all synchronisation to the user. This option is easy to implement, but is extremely error prone for the user.

We chose to synchronise only those routines where synchronisation is required for correct execution. For example, halo exchange routines do not include synchronisation statements because they are not required there. Images can do halo exchange in any order. However, for correct operation, these routines must be called by every image and only when no image is updating the cell states. This means that in practice some synchronisation must be used before and after a call to the halo exchange routine. This responsibility is left to the user. In contrast, the solidification routine uses synchronisation because it calls the halo exchange routine each iteration, and because it has to do a collective reduction operation, to determine when the model space has solidified on all images.

Using only minimal required synchronisation and avoiding single image computation are the key to a good performance.

As an example, here we analyse the performance of the solidification routine, that has a triple nested loop for all model cells in the image. Inside the loop, if a cell state is 'liquid', then it acquires the state of a randomly chosen neighbour. The loop cannot be replaced by a parallel `do concurrent` construct because it uses `random_number` intrinsic which is not `pure`, It might

be possible to parallelise the loop with OpenMP, however, we haven't explored this yet.

```
integer :: fin
logical :: finished

main: do
  array = space( :, :, :, type_grain )
  do x3 = lbr(3),ubr(3)
  do x2 = lbr(2),ubr(2)
  do x1 = lbr(1),ubr(1)
    if ( space( x1, x2, x3, type_grain ) .eq. liquid ) then
      call random_number( candidate )     ! 0 .le. candidate .lt. 1
      z = nint( candidate*2 - 1 )         ! step = [-1 0 1]
      array( x1,x2,x3 ) = space( x1+z(1), x2+z(2), x3+z(3), type_grain )
    end if
  end do
  end do
  end do
  space( :, :, :, type_grain ) = array

  finished = all( space( lbr(1):ubr(1), lbr(2):ubr(2), lbr(3):ubr(3), &
                   type_grain ) .ne. liquid )
  fin = 1
  if (finished) fin = 0

!!!  exit if (fin .eq. 0) on *all* images

end do main
```

A global reduction, over all images, is required, to calculate `sum(fin)` over all images. If it has a non-zero value, then the iterations continue and another run of the triple loop is done. If the global sum of `fin` is zero, then the model is fully solidified on all images, and the iterative process is complete.

Fig. 3 shows four reduction strategies, which we have used for the purpose of a speed-up analysis. In all cases `img = this_image()` and `nimgs = num_images()`. The easiest method, shown in Fig. 3(a), is where a single image is reading values from all other images and computes the sum. All other images wait. A single `sync all` global barrier is required in this case.

A slightly more complex strategy, shown in Fig. 3(b), is where every image adds its value to the global total, kept on image 1. This is done one image at a time with a pair of matching `sync images` statements. However, a global barrier, `sync all`, is still required to make all images wait for image 1, before they can read the updated value from it.

In both strategies (a) and (b) only one image at a time is doing the work, which is very inefficient. An improvement can be achieved with a *divide & conquer* type of algorithm (or a binary search), Fig. 3(c). This example algorithm works only when the number of images is a power of 2, i.e. `num_images()`$= 2^p$. The loop takes only p iterations. On the first loop iteration all even images add their values to lower odd images. On each following iteration the step between the images in each pair increases by a factor of 2. On the last iteration image `num_images()+1` adds its total value to that of image 1. For the case of $2^3 = 8$ images ($p = 3$) this can be schematically illustrated like this: $i = 1 : (1) \leftarrow (2), (3) \leftarrow (4), (5) \leftarrow (6), (7) \leftarrow (8)$;

$i = 2 : (1) \leftarrow (3), (5) \leftarrow (7)$; $i = 3 : (1) \leftarrow (5)$. Synchronisation is done with pairs of matching `sync images` statements. As in case (b), `sync all` is required at the end to make all images wait for image 1 before copying its total value.

The final algorithm uses `co_sum` collective, Fig. 3(d), which at this time is still a Cray compiler extension, but should become an intrinsic function when TS 18508 [6] is adopted into the Fortran standard. No synchronisation is required in this method. This is because using a collective subroutine in this case satisfies the two rules of the technical specification [6]. Rule 1: 'If it is [collective] invoked by one image, it shall be invoked by the same statement on all images of the current team in execution segments that are not ordered with respect to each other'. Rule 2: 'A call to a collective subroutine shall appear only in a context that allows an image control statement'. Hence, there is no possibility of any two images entering `co_sum` at different loop iterations. This ensures that all images exit the main loop at the same iteration count. Likewise, there is no possibility of a deadlock.

```
if ( img .eq. 1) then
  save_fin = fin
  do i = 1,nimgs
    fin = fin + fin[i]
  end do
  fin = fin - save_fin
end if

sync all
fin = fin[1]
```

(a) image 1 only

```
if ( img .ne. 1) then
  sync images ( img - 1 )
  fin[1] = fin[1] + fin
end if

if ( img .lt. nimgs ) &
 sync images ( img + 1 )

sync all
fin = fin[1]
```

(b) all images in turn with `sync images`

```
step    = 2
stepold = 1

redu: do i = 1,p
  if ( mod(img,step) - 1 .eq. 0 ) then
    sync images ( img + stepold )
    fin = fin + fin[ img + stepold ]
  else if ( mod(img+stepold,step) - 1 .eq. 0 ) then
    sync images ( img - stepold )
  end if
 stepold = step
 step = step * 2
end do redu

sync all
fin = fin[1]
```

(c) divide & conquer with `sync images`

```
call co_sum( fin )
```

(d) `co_sum`

Figure 3: Four reduction algorithms and associated synchronisation strategies used in the speed-up analysis.

Fig. 4 shows scaling of the solidification routine with the above four different implementations of the collective operations. These tests were run on a model with 2^{30} cells, using from $2^3 = 8$ to $2^{15} = 32768$ cores, i.e. with the model coarray defined from `(512,512,512)[2,2,2]` to `(32,32,32)[32,32,32]`. The times were calculated with `cpu_time` intrinsic. Each test was run 3 times, and the error bars show the fastest and the slowest times.

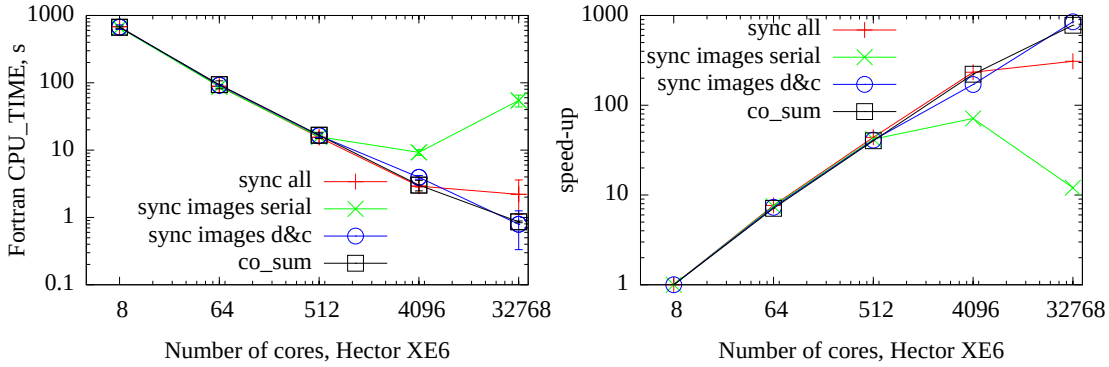


Figure 4: Timing and speed-up of the solidification routine with different implementations of the collective operation and corresponding synchronisation.

For up to 512 cores, there are no differences between the four cases. However, for higher core counts, the `co_sum` and the divide & conquer reductions show the best speed-up. For these two strategies the speed-up is nearly linear (on the log-log scale) up to at least 2^{15} cores. The speed-up is nearly 1000 for the core count raising by a factor of 2^{12}, from 2^3 to 2^{15}. We could not conduct experiments with $2^{16} = 65536$ cores due to budgetary limitations. Nevertheless, these speed-up results are very encouraging.

In addition to the scaling analysis, we have also performed profiling analysis of the solidification routines using CrayPAT API calls [2]. Profiling was done with a 2^{32} cell model run on 512 images. Tab. 1 shows the profiling results for reduction strategies (a), (c) and (d) in Fig. 3. All values, except the last row, are percentages of the total time spent in each part of the code. Note that model (d), with `co_sum` spends twice as long doing the global reduction as the triple loop computation. This is the exact opposite of strategies (a) and (c), calculation on image 1 with `sync all` and divide & conquer, where the triple loop takes roughly twice as long as the global reduction. Nevertheless, the `co_sum` approach is twice as fast overall.

	(a), `sync all`	(c), D&C	(d), `co_sum`
triple loop	50	60	25
global reduction	35	25	61
serial reduction + I/O	10	10	2
halo exchange	5	5	12
Time, s	35.0	47.1	19.7

Table 1: Relative times spent in different parts of the solidification routines and the total run time in seconds.

Another example of a global reduction operation is when calculating grain volumes: `integer, allocatable :: gv(:)[:,:,:]`. First each image calculates its local grain volume coarray

array. Then a global sum is calculated over all images. We note that using `co_sum` on 512 cores is 2 orders of magnitude faster than a single image computation.

8 Concluding Remarks and Future Work

The coarray CA microstructural library gives useful predictions of solidification, grain size distribution and cleavage fractures. Most of the code scales well up to 30k cores. A notable exception is the output routine, which is serial. The standard forbids a file to be connected to more than one image. This makes designing parallel I/O with coarrays hard. Intrinsic collectives, such as `co_sum`, lead to a better performance compared to user written collective routines. There are many triple nested loops in the library routines. It is possible that these can be optimised with OpenMP. Note that at present only Cray compiler supports OpenMP with coarrays. Finally, we are looking for a suitable scalable parallel open source finite element code to interface with the library, and ParaFEM (http://parafem.org.uk) will be our first choice.

9 Acknowledgments

This work made use of the facilities of HECToR, the UK's national high-performance computing service, which is provided by UoE HPCx Ltd at the University of Edinburgh, Cray Inc and NAG Ltd, and funded by the Office of Science and Technology through EPSRC's High End Computing Programme. This work also used the computational facilities of the Advanced Computing Research Centre, University of Bristol - http://www.bris.ac.uk/acrc/.

References

[1] I. D. Chivers and J. Sleightholme. Compiler support for the fortran 2003 and 2008 standards, revision 12. *ACM Fortran Forum*, 32(1):8–19, 2013. http://www.fortranplus.co.uk/resources/fortran_2003_2008_compiler_support.pdf.

[2] Cray Inc. *Using Cray Performance Measurement and Analysis Tools*. MAR-2013. http://docs.cray.com/books/S-2376-610/.

[3] S. Das, A. Shterenlikht, I. C. Howard, and E. J. Palmiere. A general method for coupling microstructural response with structural performance. *Proceedings of the Royal Society A*, 462(2071):2085–2096, 2006.

[4] G. Guillemot, Ch.-A. Gandin, and M. Bellet. Interaction between single grain solidification and macro segregation: Application of a cellular automaton - Finite element model. *Journal of Crystal Growth*, 303:58–68, 2007.

[5] ISO/IEC 1539-1:2010. *Fortran – Part 1: Base language, International Standard*. 2010. http://j3-fortran.org/doc/standing/links/007.pdf.

[6] ISO/IEC JTC1/SC22/WG5 N1983. *Additional Parallel Features in Fortran*. JUL-2013. ftp://ftp.nag.co.uk/sc22wg5/N1951-N2000/N1983.pdf.

[7] J. Phillips, A. Shterenlikht, and M. J. Pavier. Cellular automata modelling of nano-crystalline instability. In *Proceedings of the 20th UK Conference of the Association for Computational Mechanics in Engineering, 27-28 March 2012, the University of Manchester, UK*, 2012.

[8] D. Raabe and R. C. Becker. Coupling of a crystal plasticity finite-element model with a probabilistic cellular automaton for simulating primary static recrystallization in aluminium. *Modelling and Simulation in Materials Science and Engineering*, 8:445–462, 2000.

[9] A. Shterenlikht and I. C. Howard. The CAFE model of fracture – application to a TMCR steel. *Fatigue and Fracture of Engineering Materials and Structures*, 29:770–787, 2006.

Performance Analysis of a Large Memory Application on Multiple Architectures

Peter M. Kogge

University of Notre Dame, Notre Dame, IN, USA
kogge@cse.nd.edu

Abstract

The Graph500 Breadth-First Search benchmark has emerged as a well-documented PGAS-style application that both scales to large data set sizes and has documented implementations on multiple platforms over multiple years. This paper analyzes the reported performance and extracts insight into what are the leading performance limitations in such systems and how they scale with system size.

1 Introduction

The most common benchmark[2] to date for supercomputers has been the solution of dense linear equations of the form $Ax = B$, using the LINPACK package. The TOP500 web site[1] has been recording such measurements for almost 20 years, and ranked systems on the basis of their reported sustained floating point operations per second, denoted R_{max}. Reports submitted for consideration in the listings usually provide not only R_{max} but also R_{peak} (the peak possible flop rate) and N_{max} (the dimension of the matrix at which the measurement was made).

A major objection to the generality of the TOP500 benchmark is that it is too regular, and too floating point intensive. A second objection is that it does not factor in data set size on a comparative basis. Detailed analyzes such as [5] indicate that in real, large, scientific codes the percentage of floating point is much less than in LINPACK, with address computations and memory referencing taking center stage. This trend is expected to grow even further as dynamic grids and multi-physics become more common.

In contrast, the Graph500 benchmarks[2] are meant to stress parts of a system not key to LINPACK, such as handling extremely large data structures that must encompass many physical nodes, and high amounts of unpredictable references into these structures. The current benchmark, **Breadth First Search** (**BFS**), involves creating a very large graph and then finding all vertices that are connected to some randomly chosen root vertex. The key performance metric is not flops but "Traversed Edges Per Second" (TEPS).

The only way to solve such problems, especially for the very largest graphs, is to employ a large parallel system where the major data structures must be partitioned over many nodes and embedded links are made between different vertices on arbitrary pairs of nodes, making this an archetypal PGAS-style problem.

After three years of measurements on now 100s' of systems, the key observation is that the peak system's TEPS improved by about 3,600X since the first listing, at a compound annual growth rate (CAGR) of 62X per year for the first 2.5 years, followed by what appears to be flattening. This is in contrast to the almost monotonous 1.9X CAGR observed in the TOP500 over 20 years.

[1] www.top500.org
[2] www.graph500.org.

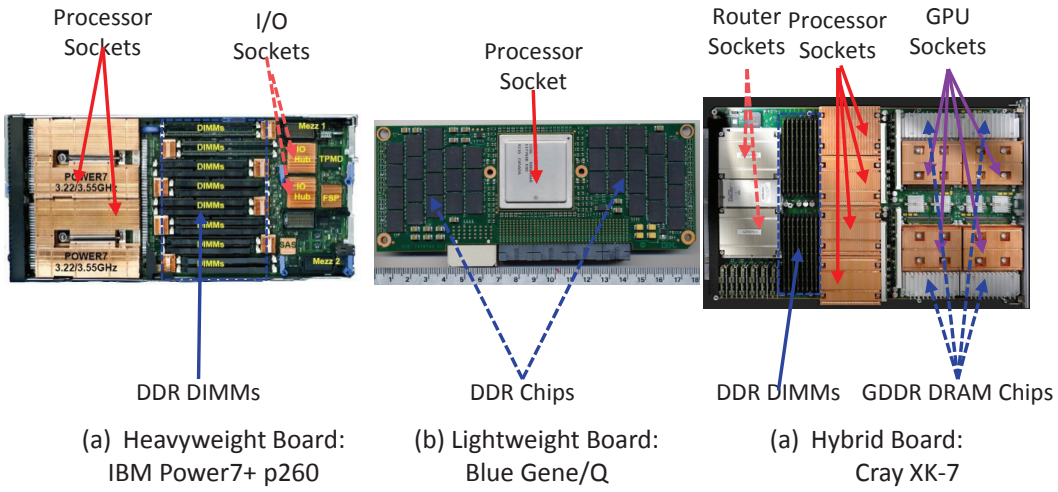


Figure 1: Typical Node Boards for different Architecture Classes.

The goal of this paper is to dive into this explosive growth rate and understand where the performance increases come from, and what class of systems are most effective in achieving them. In terms of organization, Section 2 defines the classes of architectures used in this study. Section 3 discusses the Graph500 benchmark. Section 4 discuss the range of typical implementations of the benchmark on such systems. Section 5 overviews the as-reported Graph500 results in terms of architectures that support them. Section 6 discusses how these results reflect on scalability in the underlying systems. Section 7 looks in detail on a series of implementations using two lightweight systems. Section 8 concludes.

2 Architecture Classes

In analyzing the future of supercomputers, the Exascale report[4] defined two general classes of architectures that have shown up in TOP500 rankings, **heavyweight** and **lightweight**. Since then the **hybrid** class has appeared in the TOP500. Fig. 1 pictures typical nodes for these classes.

All three classes have also appeared in the Graph500 rankings, with the addition of several more specialized, but more PGAS-relevant architectures. Each class is discussed briefly below.

In these descriptions, a "socket" refers to a high density logic chip such as a microprocessor that performs some major function, a "core" refers to a logic block capable of independently executing a program thread, and a "node" is that collection of sockets, memory, and other support logic that make up the minimum-sized replicable unit in a parallel system.

2.1 Heavyweight Architectures

Heavyweight architectures are the natural progression of what are now the ubiquitous multi-core microprocessors, and are designed to work with a combination of support chips to provide the most possible performance per chip, typically at high clock rates and with a fairly

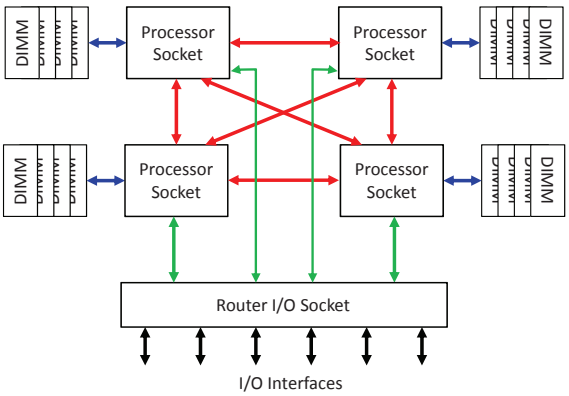


Figure 2: Modern Heavyweight Node Architecture.

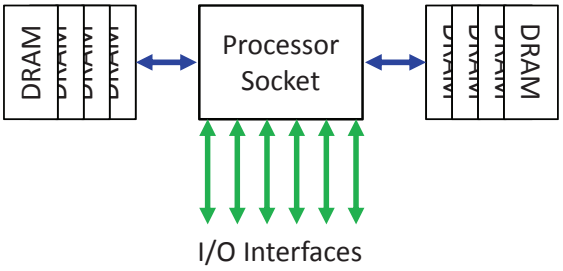


Figure 3: Modern Lightweight Node Architecture.

large amount of memory. Fig. 2 diagrams a typical heavyweight node, with Fig. 1(a)[3] a sample node from a POWER7+ system, where much of the board is taken up by the heatsinks for the processor sockets and the I/O sockets.

The chips in modern heavyweight processor sockets have 8-16 cores, each with 2-4 FPUs, and run at or above 3GHz. Typically up to 4 independent memory channels are supported, with each channel supporting high capacity, multi-rank, high bandwidth DIMMs.

All the cores in a single heavyweight processor socket share access to all the memory attached to that socket, usually including cache coherence. In many heavyweight designs this sharing of memory extends to some, usually small, number of other compute sockets in the same node. These compute sockets typically share an intra-node network. Thus a single node has a moderately large number of cores that share local node memory with each other, but has a more distributed memory interface when dealing with any core or memory elsewhere in the system, requiring software such as MPI for communication.

Typically at most a 100 or so such nodes can fit in a single rack.

2.2 Lightweight Architectures

The introduction of the IBM Blue Gene/L[6] in 2004 used a compute socket with a dual core processor chip that included a memory controller, I/O, and routing functions on a single chip.

[3]image from http://www.theregister.co.uk/2012/11/13/ibm_ power7_plus_flex_storwize/.

Parameter	L	P	Q	Q/P
Cores/node	2	4	16	4X
Core Clock (GHz)	0.7	0.85	1.6	1.9X
Max Node Memory (GB)	1	4	16	4X
Memory Ports per Node	1	2	2	same
Memory B/W per Port	5.6	6.8	21.35	3.1X
Total Memory B/W (GB/s)	5.6	13.6	42.7	3.1X
Inter-node Topology	3D	3D	5D	
Links per Node	12	12	22	4.7X
Bandwidth per Link (GB/s)	0.175	0.425	2	4.7X
Total Link B/W (GB/s)	2.1	5.1	44	8.6X

Table 1: BlueGene Family Characteristics.

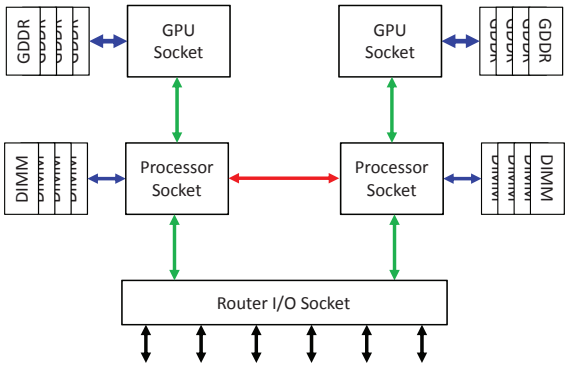


Figure 4: Modern Hybrid Node Architecture.

The cores were much simpler than for the heavyweight machines, and ran at a much lower clock rate. Such a chip, when combined with some memory, made a complete node in the above sense, as pictured in Fig. 3. Because the required heatsink was so much smaller than for a heavyweight, many of these small cards could be packaged in the same space as a heavyweight node (up to 1024 of such nodes in a single rack). Subsequent versions, Blue Gene/P[1] and now Blue Gene/Q[3], as pictured in Fig. 1(b)[4], have continued this class of architecture. Blue Gene/Q in particular has a 16+1 multi-core processor chip, with two memory controllers connected directly to conventional DDR3 DRAM chips on the same node board. Table 1 summarizes some of the key parameters of these systems.

Again all cores in a node's processor socket view the node as a single shared memory structure, but as with the heavyweight nodes, other nodes are viewed in a distributed fashion, requiring software such as MPI for communication.

2.3 Hybrid Architectures

The original Exascale report[4] identified only the heavy and lightweight classes. Since then, a third class has surfaced in the Top500, which combines with a heavyweight socket a second compute socket where the chip in it boasts a large number of simpler cores, usually with an even

[4]image from http://www.cpushack.com/wp-content/ uploads/2013/02/IBM51Y7638_BlueGeneQ.jpg

larger number of FPUs per core. Today such chips are derived from **Graphics Processing Unit** (**GPUs**), such as the Nvidia Tesla architecture[5], the Intel Xeon Phi architecture[6], or the AMD GCN architecture[7].

Fig. 1(c) pictures one such node from the Titan supercomputer[8].

As with the heavyweight nodes, something of on the order of a 100 such nodes could be packed into a single rack.

Memory within such nodes is today usually not as fully shared as in the other classes. Instead, the accelerator node typically can only access its own local GDDR (Graphics Double Data Rate) memory during computation. Thus the heavyweight host processor must explicitly transfer between the accelerator's memory and its own memory. This staging takes both time and program complexity, and derates the usefulness of the accelerator when random accesses to larger amounts of data than can fit in the GDDR is needed.

2.4 Other Architecture Classes

In addition to systems that fall into the above three classes, two other system families have shown up in the Graph500 rankings. First is the Cray XMT-2 massively multi-threaded system. This architecture is particularly relevant to the GRAPH500 problem because it natively supports a large PGAS shared memory model, when a core anywhere in the system can directly access a memory anywhere else, without intervening software.

The second system architecture consist of variations of the Convey FPGA-based systems[9]. This architecture is also particularly relevant because its memory system, albeit smaller than that possible with large clusters, has much more internal bandwidth, making it a good match again to BFS.

3 The BFS Benchmark

Several benchmarks are planned under this Graph500 umbrella, with only one of them, Breadth First Search (**BFS**), currently defined and tracked through several generations of systems. Two other benchmarks (Shortest Path and Maximal Independent Set) are planned in the near future. Unlike the scientific-orientation of LINPACK, these benchmarks are believed to be highly related to areas such as cybersecurity, medical informatics, data enrichment, social networks, and symbolic networks.

The purpose of the kernels defined in BFS is to build a very large graph, and then start at any random vertex and identify all other vertices that are connected to it.

There are three major steps in the benchmarking process:

1. Graph construction: create a data structure to be used for the BFS. The two major configuration parameters that go into this are:
   - Scale: base 2 log of number of vertices (N) in the graph.
   - Edgefactor: ratio of total number of edges to total number of vertices in the graph.

[5]http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries- Overview-LR.pdf
[6]http://www.intel.com/content/www/us/en/processors/xeon/ xeon-phi-detail.html
[7]http://www.amd.com/us/Documents/GCN_Architecture_ whitepaper.pdf
[8]http://techreport.com/r.x/ 2012_10_29_Nvidia_Kepler_powers_ Oak_Ridges_supercomputing_Titan/titan-blade.jpg
[9]http://www.conveycomputer.com/

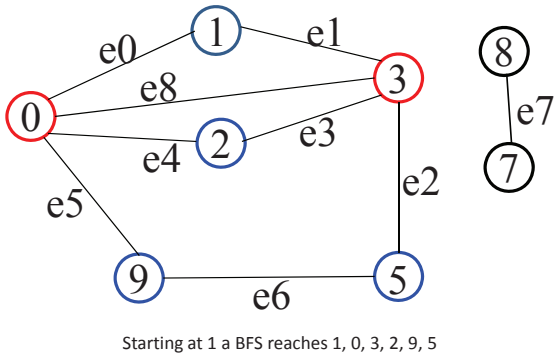


Figure 5: A Sample Graph.

Category	Level	Scale	Vertices (Billion)	Memory (TB)
Toy	10	26	0.1	0.02
Mini	11	29	0.5	0.14
Small	12	32	4.3	1.1
Medium	13	36	69	17.6
Large	14	39	550	141
Huge	15	42	4,398	1,126

Table 2: GRAPH500 Problem Size Categories.

2. Breadth-First Search: starting at a random vertex, follow all edges from that vertex to all vertices reachable over a single edge, and repeat from each vertex that had not been touched before until as many vertices as possible have been reached.

3. Validation: check that the answer is correct.

Fig. 5 shows a sample graph. Starting at vertex 0, a valid BFS output would be 0, 1, 2, 9, 3, 5. Starting at 2 a valid output would be 2, 3, 0, 9, 5, 1.

If M is the total number of edges within the component traversed by a BFS search (the second step), and T is the time for doing that search, then the key reported Graph500 metric is **Traversed Edges per Second** (**TEPS**), computed as is M/T. The time for the first and third steps is not part of the benchmark.

As with the TOP500's N_{max}, there is a problem size component to the GRAPH500, although in this case it is far more important to evaluate the success of a system than N_{max} is to TOP500. Table 2 lists different classes of problem sizes and a typical memory footprint of the resulting data structure in bytes, assuming an average node and its associated edges (on average 32 of them) take about 282 bytes. The "Level" in this table is approximately the base 10 log of the estimated size in bytes, and "Scale" is the base 2 log of the number of vertices. To date, very few systems have reached the "Large" category (needing 140TB), let alone the "Huge" category.

Note in Table 2 the scale in each problem category goes up by 3 (a growth in size of 8X) as the level goes up by 1 (a growth by a factor of 10), except between levels 12 and 13, where there is a growth by 16X in size. This extra step is to account for the difference between 8^3 and 10^3.

4 BFS Implementations

The Graph500 web-site gives reference codes for both the graph generators and the BFS algorithm itself. For the latter, several different reference codes are given, some of which are described in the subsections below.

4.1 Sequential Code

The reference sequential code (in file seq_csr.c) has several very large data structures, where N is the number of vertices ($N = 2^{scale}$, which for toy problems is 64 million and for huge problems is 4 trillion) and M the number of edges in the graph (for Graph500, $M = 32N$):

- An array *xoff* of size 2N has two words per vertex, indexing to the start and end of a list in *xadj* that includes the edges leaving that vertex.
- An array *xadj* of length M, where each word holds the destination vertex of an edge.
- An array *bfs_tree* of length N, where the v'th entry corresponds to vertex v, and gives either the index to v's " parent" vertex in terms of the BFS search or a special code if v has not yet been touched.
- An array *vlist* of length N that contains the vertices in the order in which they were touched during BFS, with vertices that were touched in the same "level" of the search arranged in sequential order.

The sequential code is a triply nested loop where:

- the innermost loop iterates over all edges exiting a vertex touched in the last level, determining if the vertices on the other side have been touched before, and if not marking them as touched in *bfs_tree* and adding them to *vlist* as part of the next level,
- the middle loop iterates over all vertices touched in the last level of the BFS search,
- the outer loop repeats the inner two as long as new vertices were added to a new level.

Overall time complexity is on the order of O(N+M).

4.2 XMT Code

The XMT reference code demonstrates the advantages of programming a PGAS problem in an architecture where all memory is visible to all threads. It is virtually identical to the sequential code with two exceptions: iterations of the inner two loops of the BFS are performed in parallel by different XMT threads, and the updates to *bfs_tree* and *vlist* are done with atomic operations that guarantee that if two threads try to touch the same vertex at the same time, only one wins.

4.3 MPI Codes

The BFS problem quickly scales to problem sizes that will not fit in the memory of any single node of any architecture class. While the XMT does scale to bigger sizes with multiple nodes, its current implementations still don't grow large enough in total capacity to match the higher

scales. Thus a third reference implementation partitions the data structures onto different cluster nodes and uses explicit MPI calls to create the equivalent of a PGAS implementation.

The approach taken in the reference codes is to partition the equivalent of *xoff* and *vlist* into approximately equal-sized pieces, one per node. *xadj* is also partitioned so that all edges for all vertices on a node are also on the same node. When the BFS code running on a node encounters an edge to a vertex that needs to be touched, the code explicitly computes on which node the vertex resides, and dispatches MPI messages to so inform the other node.

Besides the explicit computation of which node hosts a vertex, this code also handles differently the process of identifying which vertices have been recently touched, and thus are to be explored in the next level. A bit vector is kept on each node with one bit for some number of local vertices. When any node follows an edge to a vertex, the associated bit on the appropriate node is set to 1 atomically via an MPI OR sent to that node.

Two copies of these bit vectors are kept, one to record newly touched vertices and one from the last level to indicates which vertices were touched on the last level. At the end of each level, these vectors are reversed, and the one holding the previous (and now explored) markings reset to all zeros. This is to avoid the complexity of remotely enqueuing a newly-touched vertex on a remote queue.

In addition, the two states of a vertex of being touched (and having a parent), and untouched are expanded to three states: untouched, touched and explored, and touched but not yet explored. An MPI atomic MIN operation sent to the target node does both a check and state transition.

There are thus two MPI atomic operations per edge traversed: set the bit and do a MIN. Transit times and overheads for these messages greatly outweigh the execution costs at their destination, making them the primary gate on TEPS.

Finally, gluing this together is a series of barriers to synchronize all nodes between levels, and to determine when the last level has been reached and there are no more vertices to explore.

As will be seen in Section 7, over time there has been a significant increase in performance due to algorithms only. The most common approach is to sort updates to other nodes by node number, and send single packets with multiple vertex updates attached, thus amortizing overhead. Packing vertices by path may also be used so that updates that are "on the way" to some remote node can piggyback and not consume independent bandwidth.

5 Graph500 Results vs Architecture

A ranking for a system in the Graph500 provides a description of the system used and a subset of key performance parameters. The former include primarily core and node count, total memory, and a verbal description of the cores and sockets. The latter include the TEPS rate, the scale of the problem solved, and a classification of the algorithm used. For this paper, all such listings were downloaded and augmented where possible with additional information about each system, with the primary addition being a tag indicating the class of architecture into which the system falls. This tag was used to select a symbol to use on all the scatter plots that follow:

- Red squares are heavyweight systems.
- Green circles are lightweight systems.
- Purple triangles are hybrids.
- Brown stars are other.

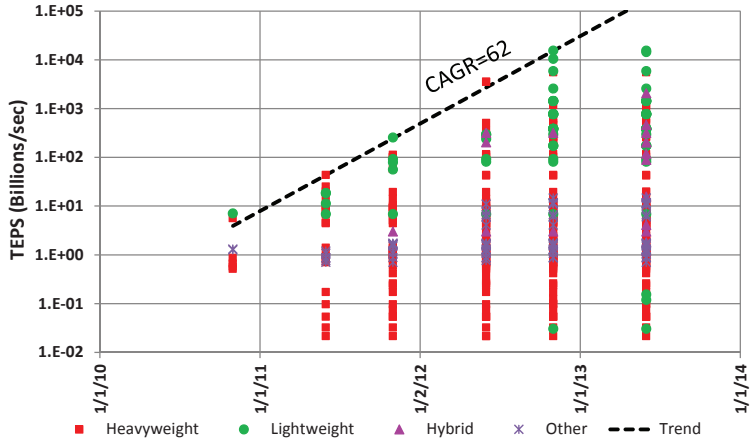


Figure 6: TEPS over time.

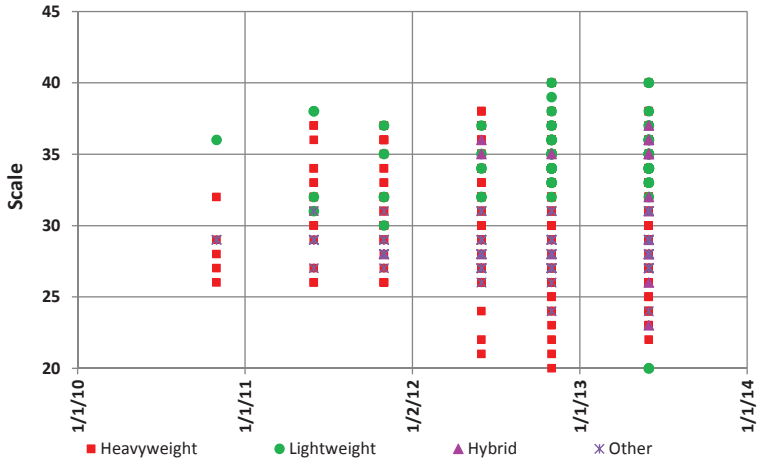


Figure 7: Scale over Time.

Fig. 6 graphs the TEPS rate over time for all reported systems. The key observation is that the peak systems' TEPS have improved by about 3,600X from the first listing in Nov. 2011 and Nov. of 2012, at a nearly constant compound annual growth rate (CAGR) of 62X per year, but this appears to have flattened as of June 2013. It is also interesting to see that until Nov. 2012 there was a noticeable mix of architecture classes near the top of the rankings, but after this, all points in the top order of magnitude are lightweight.

A second observation is the now six order of magnitude range of performance in the most recent rankings, in comparison to the only 230X difference between number 1 and number 500 for the Top500. The reason is that there is for the Graph500 a lot more interest in performance *as a function of scale*. Fig. 7 provides a time-denoted diagram of scale to match Fig. 6. While it has nowhere near the CAGR, again nearly all of the biggest problems solved by size are solved on lightweight systems.

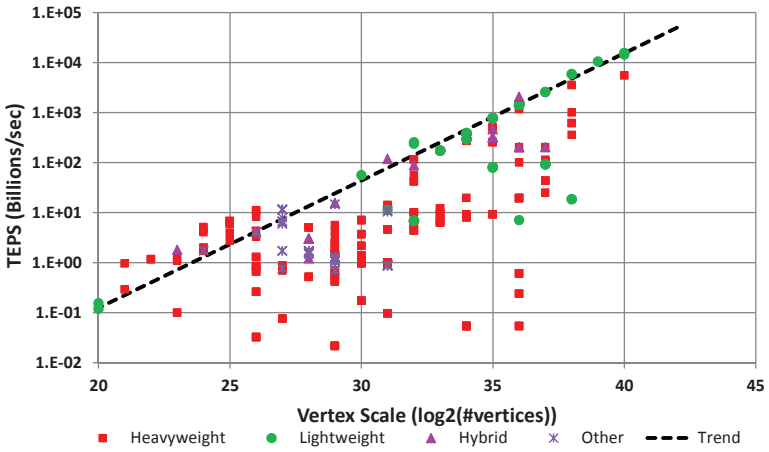


Figure 8: TEPS versus Scale.

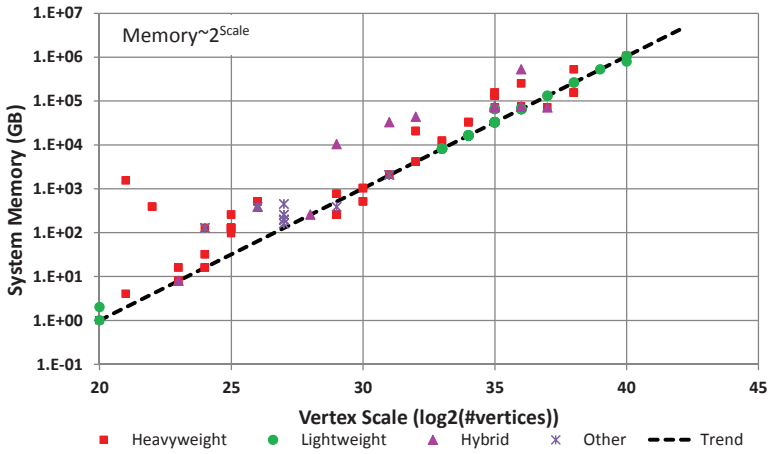


Figure 9: Memory Capacity versus Scale.

This dominance of lightweight systems is seen again clearly in Fig. 8. After a scale of 30 (1 billion vertices), lightweight systems are uniformly at the peak of the performance metric at each scale.

Also interesting in Fig. 8 is that the dotted line in this graph is proportional to 1.8 to the power of scale. Given that the number of vertices is 2 to the scale, this means that performance grows slightly more slowly than the size of the problem solved. This will be discussed more in the next section.

Fig. 9 then graphs the memory capacity of the reported systems versus the maximum size problem they solved. The close clustering of the points to the dotted line indicates that most systems did in fact try to select problem sizes that came close to filling memory.

Finally, to couple the two key metrics, TEPS and scale, Fig. 10 graphs the product of the TEPS rate and the number of vertices versus time. Peak values here are growing faster than linearly with time, indicating that both metrics are improving in concert.

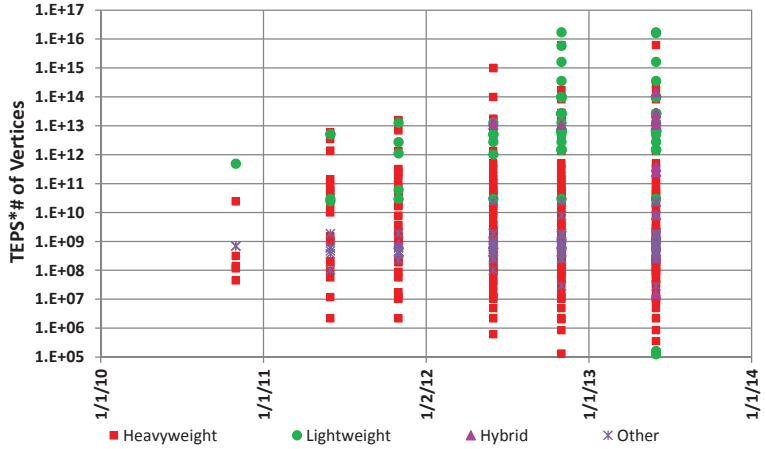


Figure 10: TEPS*Size over Time.

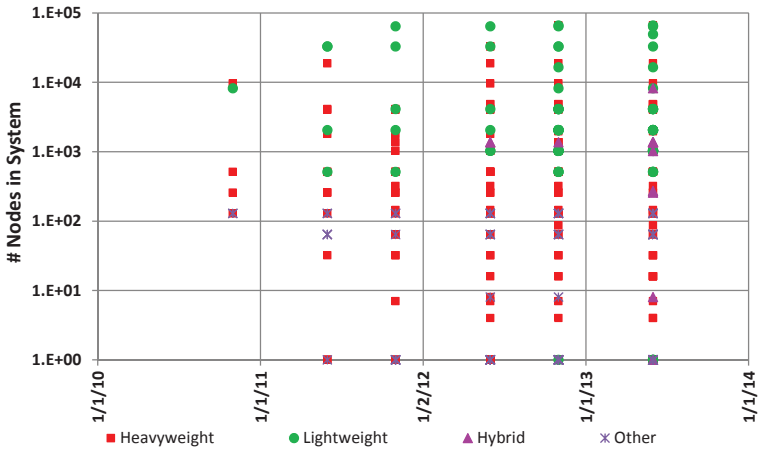


Figure 11: Node Count Growth in the Rankings.

6 Scalability in the Graph500

The key questions to be addressed in this section are "what drives the joint growth in both TEPS and scale," and "how does this vary among architecture classes."

The most obvious place to look is in the inherent parallelism in the system, starting with the number of nodes. Fig. 11 plots the growth in node count through the rankings, where we see an order of magnitude growth in peak node count over the first year, and then a flattening.

6.1 Weak Scaling

Fig. 12 graphs TEPS as a function of just this node count, ignoring all other factors such as core count, clock rates, or time of listing.

The most interesting observation is that the best of breed has a TEPS rate proportional to the number of nodes to the 0.92 power. Further, most of these are lightweight systems, with

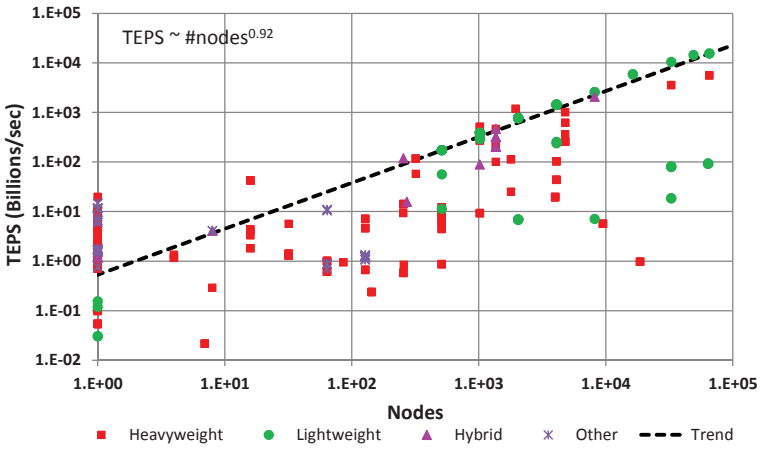


Figure 12: TEPS as a Function of Node Count.

heavyweight systems often delivering 10 to 100 times less performance at the same node count. Further, most of the leading lightweight points are BlueGene/P systems, designs with very significant amounts of inter-node bandwidth. The obvious conclusion is that, with sufficient bandwidth, BFS has almost perfect weak scaling, with the slight nonlinearity (exponent of 0.92 rather than 1) due to such things as barriers or sync points (again something for which the BlueGene has significant hardware support).

The other observation from Fig. 12 is the significant performance from single node systems. It isn't until we get in excess of 32 nodes that parallel systems outdo the best of the single nodes. This is again most probably related to bandwidth, with memory bandwidth now replacing inter-node bandwidth. This is particularly easy to see in the best of the single nodes, which are Convey systems that have a very high bandwidth partitioned memory system within them. It will be interesting to see how such systems scale when more nodes are employed, and the bandwidth between nodes becomes more conventional.

Fig. 13 is a similar graph except that it plots cores rather than nodes. This distinction expands what was the "1 node" peak before, and we see a region of single node, multi-socket, multi-core points where the system is a single shared memory system and we are replacing node-node bandwidth with multi-bank memory bandwidth.

6.2 Incremental Performance

A related way to look at the performance results is to look at the incremental growth in performance offered by each node as we grow system sizes. Fig. 14 divides the total TEPS rate of each listing by the number of nodes, and plots this versus time. This new metric provides insight into the incremental power of each additional node.

The key observation here is the over 1000X increase we see on a per node basis over the first 1.5 years, followed by an essential flattening over the last 18 months.. While some of this improvement is due to improved hardware, a more likely explanation is that the algorithms used have improved significantly (this is discussed in more detail in the next section).

The other observation is about how different architecture classes compare. In almost each ranking, the highest per node performance has been by a single node Convey box, with its large internal bandwidth. Also, the lightweight class, which provides the highest overall performance,

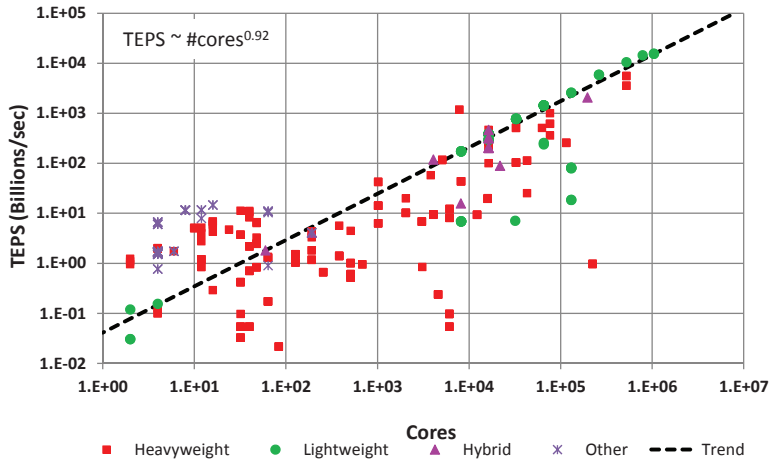


Figure 13: TEPS as a Function of Core Count.

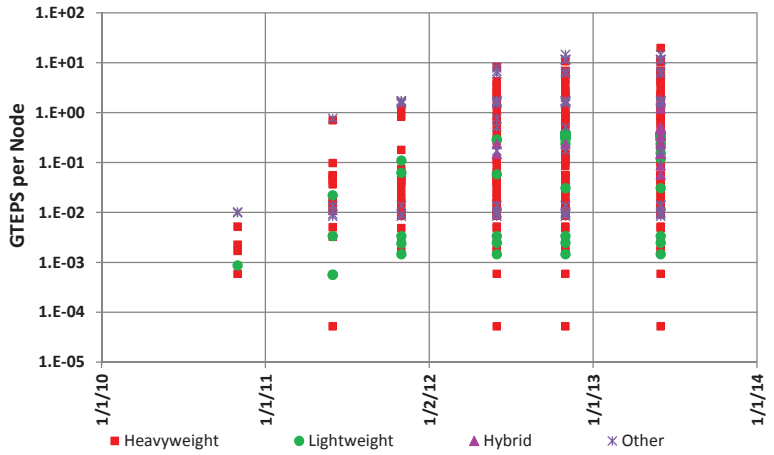


Figure 14: TEPS per Node over Time.

are middle of the pack in terms of per node performance. This is amplified by Fig. 15, which expresses the per node performance in terms of node count. Here it is obvious that the lightweight architecture becomes much more scalable as we move to larger systems.

Note also in this figure that the lightweight systems from about 1,000 to 100,000 nodes have a slight decrease in TEPS per node as node count increases. This is further evidence of the effects of serializing and decreased injection bandwidth because of cross-node traffic.

7 Detailed Analysis of Lightweight Implementations

Given the high performance of lightweight systems in these rankings, and that there are multiple systems with potentially different codes and different numbers of nodes but the same node design, a great deal can be learned by isolating on just them. In particular, we partition the

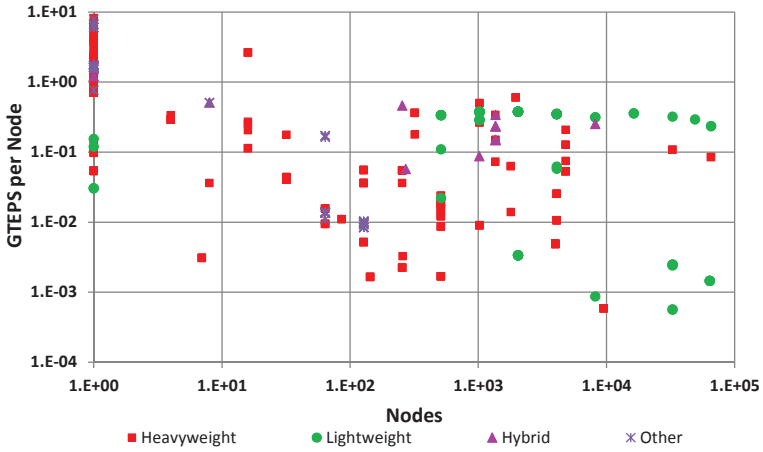


Figure 15: TEPS per Node versus Node Count.

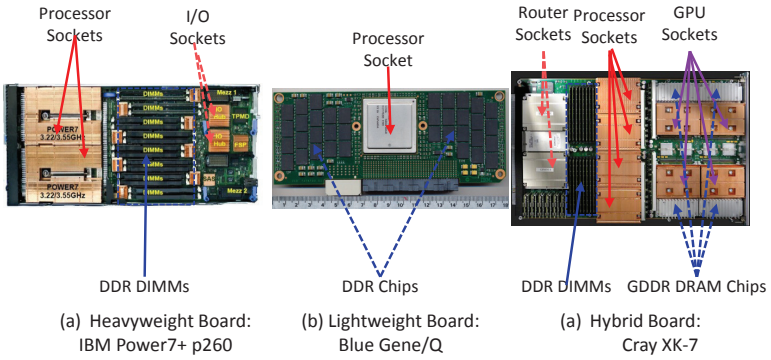


Figure 16: BlueGene TEPS per Node versus Node Count.

data points into BlueGene/P and /Q systems. Figs. 16 and 17 look at TEPS per node as a function of size and time respectively.

Fig. 17 shows that after the first ranking, the best TEPS per node for BlueGene/P was fairly constant, whereas there was about a 17X gain for BlueGene/Q before it too stabilized. Given that all the Q points have identical hardware, this increase can only be explained as optimizing the algorithms. It will be interesting to see if there are further gains in later rankings.

It is also interesting to note that there is a 112X difference between a /Q and a /P once both have reached a stable value. Earlier, Table 1 listed the hardware differences between P and Q, along with the change from P to Q. The biggest difference is in aggregate off-node bandwidth. Q has 4.7 times the number of links, with 8.6 times bandwidth per link, for a total of 40 times the total bandwidth. The difference in memory bandwidth is about a factor of 3.1. The rest must be due to better algorithms.

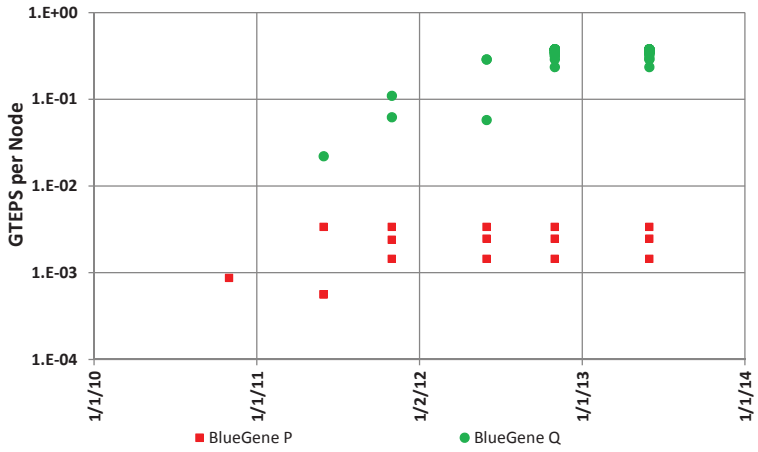


Figure 17: BlueGene TEPS over Time.

8 Conclusions

This paper has explored the results of several years of listings of performance of the BFS algorithm on a wide variety of architecture classes, where performance has two components: TEPS and graph size. TEPS performance does seem to exhibit significant performance improvements possible by algorithmic changes to optimize the use of bandwidth. Problems of small size are best processed on single-node systems with a lot of shared memory and a lot of memory bandwidth, such as found in the Convey systems. Problems of large size are best run on lightweight systems where a large number of nodes can be used. Unlike TOP500, hybrid systems are nowhere near the top in terms of performance.

In terms of details, the best systems seem to follow a near perfect weak scaling model, with an exponent of almost 1 (0.92) relating the number of nodes to performance.

Looking ahead, it will be interesting to see if performance for existing systems such as BlueGene/Q continues to improve in ways that indicate improvement in basic algorithms. Also, with the imminent release of a "Green Grap500" listing, an energy analysis akin to what has been done for LINPACK can begin. Also, it will be interesting to see if the results for BFS carry over for both the upcoming additional Graph500 benchmarks and other PGAS benchmarks. Finally, it will be interesting to see how inherently PGAS architectures such as XMT may evolve, and if they can match or exceed the performance of other architecture classes.

8.1 Acknowledgements

This work was supported in part by the University of Notre Dame and in part by Sandia National Laboratory, Albuquerque, NM under both a contract with the U.S. DoD DARPA as part of the UHPC program, and a DOE-funded extreme computing effort called XGC.

References

[1] IBM Blue Gene team. Overview of the IBM Blue Gene/P project, 2008.

[2] Jack J. Dongarra. Performance of various computers using standard linear equations software. *SIGARCH Comput. Archit. News*, 20(3):22–44, June 1992.

[3] R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Chist, and Changhoan Kim. The IBM Blue Gene/Q compute chip. *Micro, IEEE*, 32(2):48 –60, march-april 2012.

[4] Peter M. Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report CSE 2008-13, Univ. of Notre Dame, Sept. 2008.

[5] R.C. Murphy and P.M. Kogge. On the memory access patterns of supercomputer applications: Benchmark selection and its implications. *Computers, IEEE Transactions on*, 56(7):937–945, 2007.

[6] The BlueGene/L Team, T Domany, Mb Dombrowa, W Donath, M Eleftheriou, C Erway, J Esch, J Gagliano, A Gara, R Garg, R Germain, Me Giampapa, B Gopalsamy, J Gunnels, B Rubin, A Ruehli, S Rus, Rk Sahoo, A Sanomiya, E Schenfeld, M Sharma, S Singh, P Song, V Srinivasan, Bd Steinmacher-burow, K Strauss, C Surovic, Tjc Ward, J Marcella, A Muff, A Okomo, M Rouse, A Schram, M Tubbs, G Ulsh, C Wait, J Wittrup, M Bae, K Dockser, and L Kissel. An overview of the bluegene/l supercomputer, 2002.

On the Modelling of One-Sided Communication Systems

Olaf Krzikalla[1], Andreas Knüpfer[1], Ralph Müller-Pfefferkorn[1] and Wolfgang E. Nagel[1]

Technische Universität, Dresden, Germany,
{olaf.krzikalla,andreas.knuepfer,
ralph.mueller-pfefferkorn,wolfgang.nagel}@tu-dresden.de

Abstract

In the last years the one-sided communication paradigm has become important in the programming of distributed memory machines. PGAS-like APIs, and also the one-side communication facilities of MPI, have evolved significantly and now open up new opportunities for the development of HPC applications.

In this paper we present a model covering the essential features of one-sided communication systems for discussing and comparing their operational semantics. Our approach is based on task graphs, which we have extended by introducing virtual tasks. A virtual task represents an asynchronous communication operation performed by the underlying system or a RMA unit. By this means we describe the fundamental functions of three popular one-sided communication APIs, namely OpenShmem, MPI 3.0, and GASPI. We conclude the paper with an outline of a parallel algorithm based upon our model, which can be used for data race detection and performance tuning.

1 Introduction

One-sided communication systems decouple data transfer and synchronization. Their communication patterns are expected to scale better than the patterns used in two-sided communication systems, where source and destination are required to participate actively in a communication act, thus tightly coupling data transfer and synchronization.

One-sided communication systems come in a variety of flavours, ranging from PGAS languages (e.g. UPC [5]) over PGAS-like APIs (e.g. OpenShmem [6], GASPI [2]) up to extensions of existing APIs (e.g. MPI 3.0 [9]). In order to develop tools useful for a wide range of these systems models are needed, which cover the essential features of one-sided communication systems and abstract product-specific properties. Such models allow the development of generic analysis tools not confined to a particular PGAS flavour.

In this paper we introduce a model, which represents the execution of a parallel program and the sequence of memory accesses made by application program and the underlying one-sided communication system. Section 2 explains the foundation of our model: the concept of task graphs. Fundamental functions of one-sided communication systems are described in terms of our model in section 3. We draw references to specific functions of popular APIs, namely OpenShmem, MPI 3.0, and GASPI. In order to keep this section straight to the point we omit very specific features like MPI windows. Going one step further we describe in section 4, how our model can be used to reason about the correctness of memory accesses with respect to data races as well as for performance tuning. The explained algorithm works in parallel by design in order to be applicable on large systems. Eventually we give an overview of other approaches used for the modelling of parallel program executions in section 5.

2 Task Graphs

A *task graph* represents the ordering of significant events during a program execution in a directed acyclic graph. The nodes of the graph are the events. The directed edges model a happens-before relation [10] and specify guaranteed run-time orderings between the events. Due to the transitivity of this ordering a task graph provides good means to reason about a program execution, e.g. to identify potential concurrent events: if there is no path from one node to the other one in either direction, then the two corresponding events can occur simultaneously. A classic use-case is data race detection: if the events are memory accesses to the same location and at least one of them is a write access, then this represents a data race.

2.1 Virtual Tasks

A *task* can be seen as an execution path along the edges through the nodes of the graph. A task can be a process, a thread, or something different. We believe, that this definition is insufficient for the modelling of one-sided communication systems. Therefore we refine it by differentiating between *real tasks* and *virtual tasks*.

We call a task a *real task*, if its execution path represents one particular process or thread. This corresponds to the classical definition where a task graph consists of sequentially executed nodes and edges connecting real tasks and representing synchronization relations [3, 7, 11, 13, 14, 16]. This model is sufficient for programs at shared-memory systems, since memory accesses are synchronous with respect to the executing real task. However, one-sided communication systems use asynchronous memory accesses. For the application it is usually intransparent, by which means they are actually executed – it might be an underlying system process or a hardware-based facility (e.g. an RDMA unit). Therefore asynchronous memory accesses cannot be correlated to a particular process or thread and thus not to a real task. The edges to and from such a node are defined by the operations causing and handling the corresponding asynchronous memory access. We call a path through nodes, which do not belong to a real task a *virtual task*.

Virtual tasks are always forked from nodes of a real task. Nodes of a virtual task can fork further virtual tasks or act as join nodes for other virtual tasks. However they can neither fork nor join a real task. Edges between nodes of a virtual task specify a guaranteed run-time ordering in the same way as other edges do. Eventually a virtual task will be joined by a node of a real task. The joining task may be different from the forking task. In that case the virtual task forms an implicit synchronization relation between the two real tasks.

Our model does not restrict a virtual task to memory accesses. Some one-sided communication operations introduce other nodes, which might be part of a virtual task. In some cases even artificial nodes must be used in order to model the correct run-time ordering of events.

3 Modelling One-Sided Communication Operations

In the following we describe the model for every operation in two ways – with an example graph and with formal building rules. Our starting point is a task graph of all processes of the application processed in temporal order. Such a graph consists of real tasks only without any edges between them. This task graph is complemented by applying the following two types of building rules:

Node addition $\langle X \rangle \mapsto \langle Y, Z \rangle$**:** Replace the initial node on the left with the set of new nodes on the right. The first new node replaces the initial node – often it will be equal to the initial one.

Edge addition $X \to Y$**:** Introduce an edge from X to Y. The type of the edge is not further specified. Usually it will be edges from/to newly added nodes.

Memory access nodes are labeled with R_L and W_L denoting read and write accesses to local memory or R_R and W_R denoting read and write accesses to remote memory. Nodes accessing remote memory asynchronously are part of a virtual task. Nodes representing operations such as function calls are labeled with the appropriate initial letter. Some labels are augmented by superscripted letters, which represent arguments. A superscripted r is used for the target rank, other abbreviations are explained on the fly.

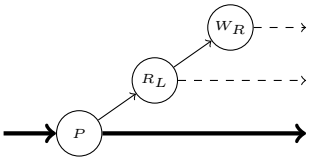
The creation rules are applied to the nodes in their actual execution order. If a node is prefixed with the universal quantifier $\forall$ in an edge addition rule, then edges are drawn from all nodes of the given type, which were already created at that time.

Beside the formal building rules examples are given illustrating the resulting graphs. For a better understanding we will depict the sequential ordering of events belonging to the same real task by bold edges. Virtual tasks execute along normal edges. Dashed edges indicate an ongoing virtual task, which will be joined to a real task at a later point.

3.1 Put and Get operations

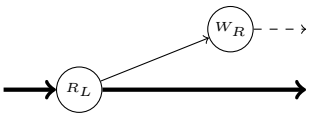
Put operations copy data from a local source memory area to a memory area of a remote target. The data is transparently written to the target – no task at the remote side needs to receive the data explicitly. Furthermore, all APIs covered in this paper perform the write operation asynchronously to the calling task. Thus we model this behavior by creating two virtual tasks from the *Put* call P, one with the local read access R_L and one with the remote write access W_R. We add an edge from R_L to W_R due to the corresponding data dependency.

The remote write access W_R is always asynchronous. In the general case the local read access R_L is asynchronous too, as shown in Fig.1, top. This is the semantic of `gaspi_write` and `MPI_RPUT`. If R_L is synchronous, it coincides with P and only one virtual task is needed. This is the case for `MPI_PUT` and all OpenShmem put routines. It is shown in Fig.1, bottom.



`gaspi_write, MPI_RPUT:`

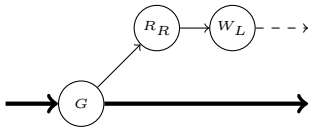
$$\begin{aligned} \langle P \rangle &\mapsto \langle P, R_L, W_R \rangle \\ P &\to R_L \\ R_L &\to W_R \end{aligned}$$



`shmem_*_put, MPI_PUT:`

$$\begin{aligned} \langle P \rangle &\mapsto \langle R_L, W_R \rangle \\ R_L &\to W_R \end{aligned}$$

Figure 1: Task graphs for *Put* operations



$$\begin{aligned} \langle G \rangle &\mapsto \langle G, R_R, W_L \rangle \\ G &\to R_R \\ R_R &\to W_L \end{aligned}$$

Figure 2: Task graph for an asynchronous *Get* as defined by `gaspi_read` and `MPI_RGET`.

Get operations copy the data from a remote memory source to local memory. This results in a remote read access and a subsequent local write access. The `shmem_*_get` routines and the `MPI_GET` routine are completely synchronous operations. A synchronous *Get* returns once the data has been copied to the target address. Thus it does not introduce additional virtual tasks or nodes and the corresponding rule is omitted due to its simplicity. In contrast an asynchronous *Get* as defined by `gaspi_read` and `MPI_RGET` issues the copy request to the underlying system and returns directly afterwards. This not only delegates the remote read R_R to a virtual task, but also enforces the local write W_L to that virtual task due to the data dependency of W_L on R_R (Fig.2).

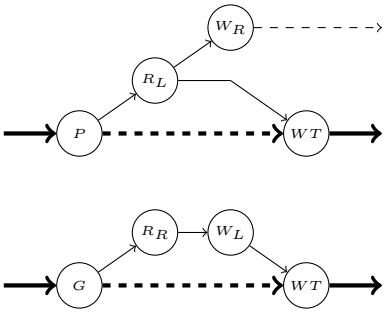
3.2 Local Synchronisation

The *Put* and *Get* operations defined by MPI and GASPI invoke asynchronous accesses to local memory. Hence both APIs provide *Wait* operations in order to synchronize these accesses with the local task. In the task graph model a *Wait* operation is always a pure join node. It does not introduce other nodes but only edges from previous nodes to itself.

GASPI provides message queues in order to synchronize operations. The function `gaspi_wait` takes a queue number q as an argument and blocks until all *Put* and *Get* operations posted to that queue have completed their accesses to local memory. Therefore, the formal notation given in Fig.3 covers all previous operations for queue q.

MPI provides the `MPI_Wait` and `MPI_Waitall` functions in order to wait for local completions of one or multiple requests *req*. Thus – unlike GASPI – a MPI *Wait* targets dedicated *Put* and *Get* operations. In the formula given in Fig.3 the request *req* determines, whether a read or write access is targeted.

As can be seen from the graph illustrating the *Get/Wait* interaction in Fig.3, bottom, a *Get* operation is completed when the associated *Wait* returns. This includes both the local write access and the remote read access. For a *Put* operation the situation is different. The state of the remote write access W_R is irrelevant for the local *Wait* operation, see Fig.3, top.



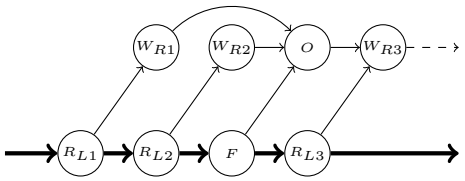
GASPI:

$$\begin{aligned} \forall\, R_L^q &\to WT^q \\ \forall\, W_L^q &\to WT^q \end{aligned}$$

MPI:

$$\{W|R\}_L^{req} \to WT^{req}$$

Figure 3: Task graphs for the *Wait* operation WT of GASPI and MPI.



$$
\begin{aligned}
\langle F \rangle &\mapsto \langle F, O^1 \dots O^n \rangle \\
F &\to O^1 \dots O^n \\
\forall\, W_R^r &\to O^r \\
\\
\langle P^r \rangle &\mapsto \langle R_L, W_R^r \rangle \\
R_L &\to W_R^r \\
\forall\, O^r &\to W_R^r
\end{aligned}
$$

Figure 4: Task graphs for subsequent OpenShmem *put* operations with a *fence* operation in-between. All *puts* have the same target rank.

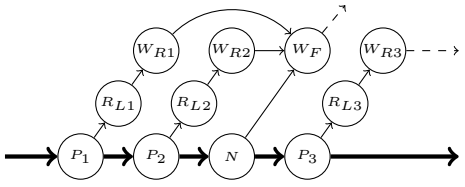
3.3 Message Ordering

The three covered APIs use two different concepts to put remote write accesses in a particular order at the receiving side. OpenShmem and MPI use the well-know approach of fences, while GASPI uses the queue approach. Interestingly we will see right below, that the two concepts form very similar task graphs.

The *Fence* operation of OpenShmem and MPI ensures, that all put requests before that operation and to a particular target rank are finished before any put request to that rank after the fence. We model a *Fence* operation by introducing an artificial *Order* node O, which is created by the *Fence* call (Fig.4). All previous remote write operations W_R^r join to O^r. Furthermore, all subsequent remote write operations W_R^r have to have edges from O^r – since they cannot be added at this point, the rules for the *Put* operation need to be refined accordingly. Note, that `shmem_fence` does not take any parameter, thus affecting *Puts* to all remote ranks. Therefore, *Order* nodes for all remote ranks are added.

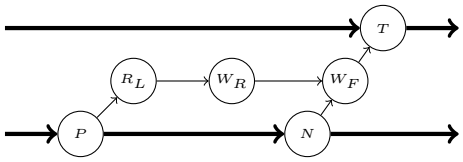
GASPI uses the queue concept for message ordering within every queue. The `gaspi_notify` routine writes a flag to the remote side over a selected queue. Like a fence it guarantees that it is remotely written after all previous *Put* operations However, GASPI does not define a relation between a `gaspi_notify` and subsequent *Put* operations, since the notification message is already part of `gaspi_notify`. Thus the task graph of `gaspi_notify` (Fig.5) differs in the following points from the task graph of `shmem_fence`:

- the ordering node O becomes the notification node W_F writing the flag.
- the fence effect is restricted to write accesses posted in the same queue q.
- there is no edge to following *Put* operations, thus their rules don‘t need to be refined.



$$
\begin{aligned}
\langle N^{r,q} \rangle &\mapsto \langle N^{r,q}, W_F^{r,q} \rangle \\
N^{r,q} &\to W_F^{r,q} \\
\forall\, W_R^{r,q} &\to W_F^{r,q}
\end{aligned}
$$

Figure 5: Task graphs for GASPI *put* operations with a *notify* operation in-between. All operations are posted to the same queue and have the same target rank.



$\texttt{Matching}(W_F) \rightarrow T$

Figure 6: Interaction of a `gaspi_write`(P), a `gaspi_notify`(N) and a `gaspi_notify_wait`(T) operation. P and N are posted to the same queue and have the same target rank, at which T is executed.

3.4 Remote Synchronisation

One-sided communication procedures manage the entire communication on the active tasks. Still, explicit synchronization is sometimes required and thus offered by the one-sided APIs. OpenShmem and GASPI implement it with a dedicated *Test* operation T which blocks until a local memory field (flag) is modified by an remote access W_F from another task. Strictly speaking this will only synchronize T and W_F but it can be combined with the message ordering mechanisms, see Sect. 3.3 above.

In the resulting task graph a *Test* operation forms a join node T at the remote task. Edges from the operation writing the flag W_F which is triggered by a *Notify* call to T are added (Fig.6). The predicate `Matching` returns all W_R, which might potentially trigger T. In order to identify those nodes, we use the algorithms given in [7]. These algorithms are applied as a last step in our task graph buildup. It must be noted that the algorithms compute only happens-before relations and thus may introduce redundant edges. In order to find exactly matching $\{W_R, T\}$ pairs a further refinement of the algorithms will be needed.

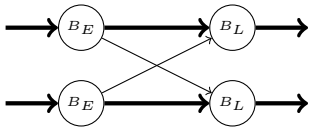
3.5 Collectives

Collective operations are not a specific feature of one-sided communication APIs. However they provide important synchronization means and must be modeled in the task graph. Here we consider only blocking collectives, ignoring the nonblocking collectives from MPI 3.0.

A group of participating tasks has to enter and stay in a blocking collective before any task is allowed to leave it. The groups are defined in different ways. GASPI and MPI have their group and communicator concepts while OpenShmem calls expect the participants as a *(first:stride:last)* triple.

Yet, the task graph for blocking collective operations can be modeled in the same way for all three APIs. A pairwise interconnection of matching barrier calls is added to the task graph (Fig.7). The `matching` predicate yields the set of all matching barrier calls of remote tasks – they are the first barrier calls in the participating tasks not yet connected to their counterparts.

Reduction routines are a special form of blocking collectives. Their synchronization model is the same as for barriers. However they access memory during their execution. We model this by inserting access nodes to local memory between the B_E and B_L events.



$B_E \rightarrow \texttt{matching}(B_L)$

Figure 7: A task graph for barriers connecting *Enter*(B_E) and *Leave*(B_L) events.

Figure 8: Task graphs for lock operations. On the left side the lock semantic of OpenShmem and on the right side the lock semantic of MPI 3.0 using `MPI_RPUT` is modeled.

3.6 Locks

The locking mechanism as known from shared memory models is provided by OpenShmem and MPI 3.0. Both APIs support a *lock* operation (`MPI_WIN_LOCK, shmem_set_lock`) and an *unlock* operation (`MPI_WIN_UNLOCK, shmem_clear_lock`). And both APIs require, that matching calls to *lock* and *unlock* are made by the same process. However the semantics of locks are different in OpenShmem and MPI 3.0.

In OpenShmem a lock protects the local read access only. As can be seen in the task graph (Fig.8, left side), the write access to the remote memory is not protected by the lock. In order to protect the write access as well the programmer must construct a synchronization path starting after the *Test* at the remote task and joining to the local task before the call to *unlock*. There is no implicit relation between the local *unlock* call and the operations at the remote side.

MPI 3.0 extends the semantic of *unlock* and guarantees a completion of the remote access when *unlock* returns (Fig.8, right side). This solves the problem of the unprotected remote access. On the other hand this is a very strict synchronization. Normally a calling task should not be interested in the completion of the issued one-sided communication on the remote side. The *unlock* semantic of MPI 3.0 is somewhat contrary to that intention, since it requires some form of acknowledgment from the remote side.

Apparently the usage of the classic locking approach as a means to protect memory accesses in a one-sided communication API raises some technical difficulties. In a shared memory parallel program each read and especially each write is synchronous to its calling task. However this fact does not hold any longer for one-sided communication calls. The virtual tasks introduced by those calls create memory accesses which are asynchronous to the calling task. Locks on the other hand remain synchronously bound to their calling task thus making it difficult to provide an appropriate lock semantic for one-sided communication APIs. Consequently, GASPI does not provide a locking functionality at all. And, while locking can be incorporated in our model, we do not consider it further in this paper.

3.7 Atomic Operations

Atomic operations are provided by OpenShmem and GASPI. Since both APIs define a synchronous execution model for them, they don't introduce virtual tasks. Thus memory accesses to the location of the atomic variable are synchronous to the corresponding real task. As with synchronous *Get* operations a task graph generation rule is omitted due to its simplicity.

Nevertheless atomic operations can contribute to a task graph, if they are used for communication purposes. In this case they may introduce happens-before relations and thus additional edges between real tasks. Regrettably there is no universal rule to compute these edges as this heavily depends on the particular code. However, the design of a tool based on our model needs to take into account that there might be non-computable happens-before relations and consequently ought to provide a way to enable the user to add such relations.

4 Analysis Tools for One-Sided Communication Systems

The rules introduced in the previous section enable us to construct a complete task graph of a particular program execution. Such a task graph can reveal various properties of that program execution. For instance, a node, which is part of a virtual task, has an interesting run-time ordering in relation to a particular real task. Let us assume two nodes S, F, which are part of the real task and N being the node of the virtual task. S is defined as the last node (with respect to the sequence of the real task), from which there is a path to N. In turn, F is the first node, so that there is a path from N to F. Then from the perspective of the real task N is virtually executing during the complete time span starting at S and finishing at F. Now, if N represents a writing memory access and there is also an access between S and F to the same memory location, then we have a data race. We can even extend this analysis in order to obtain performance tuning hints. A tool can spot premature or superfluous synchronisation nodes with respect to data-race correctness by finding the largest possible span between S and F.

However the creation of such tools is challenging. As mentioned in the introduction one-sided communication systems are dedicated to be used on large systems. Therefore the algorithmic design have to take into account the expected vast amount of analysis data a priori. In the following we assume that there is a dedicated *analysis site* for every real task.

4.1 Obtaining the task graph

Generally speaking, the initial task graph can be obtained by either static or dynamic analysis. A static analysis tool would try to predict the initial task graph by analysing the program code. For small sample programs this might be a feasible approach, but an exhaustive static program-flow and data-flow analysis of a real-world application is not an option according to experience.

A dynamic analysis traces all events of interest. For the purpose of the tools described in this paper these are calls to communication operations and local memory accesses. Tracing memory accesses is a severe task especially in a HPC environment. The vast amount of trace data must be compressed in order to become manageable. Discussing compression techniques in detail goes beyond the focus of this paper. Here we only assume, that we have merged subsequent local memory accesses without intervening communication operations to one event and that we store the access intervals in the corresponding node.

The traced events form the initial task graph. The construction of the complete task graph is then performed in parallel by the analysis sites. Created nodes such as remote memory accesses remain on the creating analysis site. Communication between sites is only necessary in order to connect nodes representing barriers or *Write* and *Test* nodes. Eventually we have obtained a possibly distributed task graph of our program execution.

4.2 Data-Race Detection

We use the already outlined time span algorithm in order to find data races. The following algorithm computes the corresponding S, F pairs for every memory access.

1. Nodes in the local task graph are assigned a local incrementing time stamp starting at 0. This time stamp is considered to be the starting and finishing time stamp
2. Iterating from the last to the first node, the time stamps are recursively propagated as starting time stamps to nodes, which fork from the processed node. The propagation

stops at nodes already having a starting time stamp (which is greater or equal due the backward iteration).

3. Iterating from the first to the last node, the time stamps are recursively propagated as finishing time stamps to nodes, which join to the processed node. The propagation stops at nodes already having a finishing time stamp (which is lower or equal due the forward iteration).

4. Nodes representing remote memory accesses are sent to the site actual processing the corresponding real task graph. Each such node should have a starting and finishing time stamp. A missing starting time stamp means it starts at 0. A missing finishing time stamp means it lasts until the end of the program.

Eventually each analysis site can run the data-race detection for its assigned task graph locally and in parallel. This is done by testing for overlapping access regions in the two-dimensional time-stamp/memory-address space. An overlap is reported as a data race, if it contains at least one write access and if at least one access is not caused by an atomic operation.

4.3 Performance Tuning

The time-stamped task graph created in the previous section can also be used to obtain performance tuning hints. For this purpose it checks in a first step to which degree starting and finishing time stamps of remote memory access regions can be lowered or raised respectively without causing conflicting overlaps. It then reverse-propagates the changed stamps across the graph until a node in the local task graph (from which the time stamp originated) is reached. The changed time stamp is a possible position of the node in the task graph. If there are equal nodes skipped due to the movement of the processed node (e.g. $Wait$ operations on the same GASPI-queue), then the node could be removed. However changing the task graph in that way may affect other synchronization relations. Therefore in a second phase the changed task graph must be checked against data races over again. The propagation steps need to propagate only changed time stamps thus reducing the area of changes in the graph. Eventually the analysis has computed a set of operations, which can be postponed or removed (e.g. local $Waits$) or brought forward (e.g. $Puts$).

Our task graphs open up more opportunities to deduce performance tuning hints. For instance it is possible to identify superfluous writes (if there is no read access in-between two writes). However, discussing all these opportunities goes beyond the scope of our paper. We have provided the algorithmic sketches here primarily in order to demonstrate the usefulness of our task graph model with respect to the tool development in the realm of one-sided communication systems.

4.4 Visualization

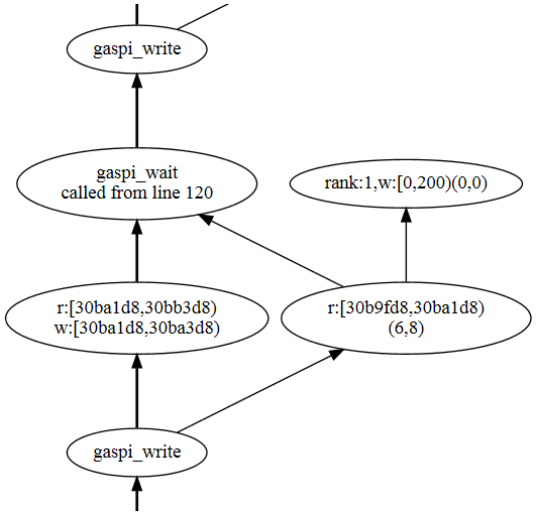
Based on the time span algorithm described above we have implemented two tools to visualize task graphs and memory access patterns. We trace local memory accesses and function calls to the GASPI API using Pin [1]. Currently we record only the effective address of every access and arguments of function call events, but no corresponding time stamps. Thus the tools can only show the logical interaction of synchronous and asynchronous memory accesses, but not yet the real-time behavior.

A simple example is shown in Figure 9. The code on the upper left performs a matrix multiplication ($m1 = m1 + m2 * m3$) and sends the result ($m1$) in slices to a remote node. After

every fourth row, the just computed four rows are sent to a remote node via a *Put* operation (in particular a `gaspi_write` call). In addition there is a *Wait* operation (`gaspi_wait`) before every *Put* thus limiting the asynchronous read access of the former *Put* call to that point in time. On the upper right side of the figure a detail of the generated task graph is shown. The task graph view combines subsequent synchronous local memory accesses in one node (represented by the `r/w` node between the `gaspi_write` and `gaspi_wait` node). The task graph view can be leveraged to spot communication patterns and relations between real and virtual tasks.

The diagram view provides a more detailed insight in the memory access events of a par-

```
// MSZ...matrix size
double* ptr;
gaspi_segment_ptr(SEG_ID, (void**)&ptr);
double *m1 = ptr,
       *m2 = ptr + MSZ*MSZ,
       *m3 = ptr + 2 * MSZ*MSZ;
unsigned rowBytes = MSZ*sizeof(double);
for (int k = 0; k < MSZ; ++k) {
  for (int j = 0; j < MSZ; ++j) {
    double value = m1[j + MSZ*k];
    for (int i = 0; i < MSZ; ++i) {
      value += m2[j + MSZ*i] * m3[i + MSZ*k];
    }
    m1[j + MSZ*k] = value;
  }
  if (k > 0 && k % 4 == 3) {
    gaspi_wait(0, GASPI_BLOCK);
    gaspi_write(SEG_ID, (k-3) * rowBytes, 1,
                SEG_ID, 0, 4 * rowBytes, 0,
                GASPI_BLOCK);
  }
}
```



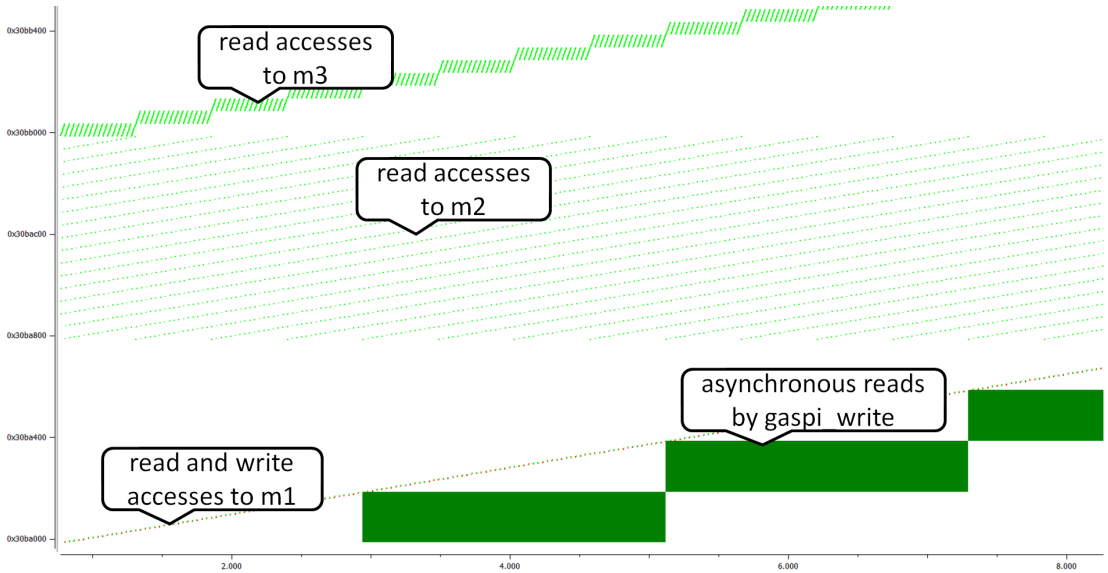


Figure 9: Visualization of memory accesses in a compact graph and a detailed diagram

ticular node. The x axis represents logical time steps, the y axis the address space. Every synchronous local memory access represents a logical time step and is visualized by one single dot according to its address. The one-strided accesses to $m3$ and row-strided accesses to $m2$ can be seen easily. Asynchronous memory accesses are visualized by rectangles. The length of a rectangle is computed by the time span algorithm and the height results from the accessed memory range of the corresponding operation. Since we do not record real time stamps yet, the *Wait* operation is not visible as a particular time step, but just limits the length of an asynchronous read access according to our model. In a future version we will trace time stamps too, thus being able to spot performance bottle necks like time-consuming *Wait* operations.

5 Related Work

Various models exist to describe and investigate the execution of parallel programs. Task graphs were introduced by Emrath et.al. in [7, 8]. This model is widely used since it represents the happens-before relation [10] in an intuitive way. Netzer et.al. in [11] extended the model by distinguishing between temporal ordering and data dependence ordering and by integrating semaphores. The resulting model was used to reason about program executions in shared-memory systems. Consequently, tools built around these models have put their focus on lock-based synchronization. The lockset algorithm was introduced by Savage et.al. in [17]. Their tool *Eraser* efficiently detects data races in lock-based programs.

Reynolds developed *separation logic* as another means to describe parallel program executions [15]. It is used by Botincan et.al. [4] in order to reason about program executions in a distributed-memory system. They introduce a *pend* assertion, that is very close to our virtual task concept. However they have focused their work on a C-like core language and they model only a small set of operations of one-sided communication systems (*Put, Get, Wait/Test*). Yet another model is the Dynamic Program Structure Tree introduced by Raman et.a. [14]. There it is applied to shared-memory systems. Notably their data-race detection algorithm works in parallel by design too.

The first data-race detection tool dedicated to PGAS-like distributed memory systems was published by Park et.al. in [13]. Their tool UPC-Thrille is implemented for the UPC programming language. In the aforementioned work the tool did not detect races due to local memory accesses yet. Accordingly they have extended UPC-Thrille and present performance results of exhaustive local memory access tracing in [12].

Bugs due to data-races are still eminent and hard to find. Thus there are plenty of other tools not mentioned here and surely plenty of tools currently in development. A rather comprehensive overview of the current state of the art can be found in [12].

6 Conclusion

The main contribution of our paper is the coalescing of the task graph model and the one-sided communication paradigm in one general approach. We have showed that our approach can be used to model the fundamental operations provided by popular one-sided communication APIs.

We consider our model as a junction point. It abstracts from the functionality of concrete one-sided communication APIs. On the other hand it can act as a starting point for the programming of various tools based on task graphs. We have outlined two possible applications in section 4.

The presented work can be extended in various ways. For each of the discussed one-sided communication APIs a completion of the description covering the respected functionality might be done. Of course, other APIs or languages using one-sided communication primitives can be described in terms of our model as well. Furthermore it is necessary to address hybrid parallel systems, which we have excluded in this paper. For this purpose the algorithm finding remote synchronisation pairs must be refined.

Our very next step is the development of the outlined data-race detection tool. In a first step this tool will reason about dynamically traced task graphs consisting of local memory accesses and calls to GASPI functions. We will use our visualization tool to spot detected data races. However the tool chain will not be restricted to the GASPI API. The abstraction layer described in this paper is built into the tool chain and makes the used one-sided communication API easily exchangeable.

Acknowledgments This work has been funded by the German Federal Ministry of Education and Research within the national research project GASPI (01 IH 11007) [2].

References

[1] Pin - A Binary Instrumentation Tool. Website. Available online at `http://www.pintool.org`; visited on June 7th 2013.

[2] GASPI - Global Adress Space Programming Interface. Website, 2011. Available online at `http://www.gaspi.de`; visited on January 26th 2013.

[3] K. Audenaert and L. Levrouw. Space efficient data race detection for parallel programs with series-parallel task graphs. In *Parallel and Distributed Processing, 1995. Proceedings. Euromicro Workshop on*, pages 508 –515, jan 1995.

[4] Matko Botincan, Mike Dodds, Alastair F. Donaldson, and Matthew J. Parkinson. Safe asynchronous multicore memory operations. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 153–162, Washington, DC, USA, 2011. IEEE Computer Society.

[5] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene D. Brooks III, and Karen Warren. Introduction to UPC and Language Specification. 1999.

[6] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.

[7] P.A. Emrath, S. Ghosh, and D.A. Padua. Detecting nondeterminacy in parallel programs. *Software, IEEE*, 9(1):69 –77, jan. 1992.

[8] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Event synchronization analysis for debugging parallel programs. In *In Proceedings of Supercomputing '89*, pages 580–588, 1989.

[9] T. Hoefler. New and old Features in MPI-3.0: The Past, the Standard, and the Future, Apr. 2012.

[10] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[11] Robert Netzer Netzer and Barton P. Miller. Detecting data races in parallel program executions. In *In Advances in Languages and Compilers for Parallel Computing, 1990 Workshop*, pages 109–129. MIT Press, 1989.

[12] Chang Seo Park. *Active Testing: Predicting and Confirming Concurrency Bugs for Concurrent and Distributed Memory Parallel Systems.* PhD thesis, EECS Department, University of California, Berkeley, Dec 2012.

[13] Chang-Seo Park, Koushik Sen, Paul Hargrove, and Costin Iancu. Efficient data race detection for distributed memory parallel programs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 51:1–51:12, New York, NY, USA, 2011. ACM.

[14] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 531–542, New York, NY, USA, 2012. ACM.

[15] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[16] Concepcio Roig, Ana Ripoll, Miquel A. Senar, Fernando Guirado, and Emilio Luque. Exploiting knowledge of temporal behaviour in parallel programs for improving distributed mapping. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismueller, editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 262–271. Springer Berlin Heidelberg, 2000.

[17] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

Coarray C++

Troy A. Johnson

Cray Inc.
troyj@cray.com

Abstract

Years ago, the C language made strides into the Fortran-dominated field of high-performance computing (HPC). Recently, a similar trend is that C++ is being used instead of C to write HPC applications. Although C and Fortran programmers have the option to use a partitioned global address space (PGAS) model via Unified Parallel C (UPC) and coarrays, respectively, C++ programmers do not have an equivalently good option. They must resort to writing a mixed-language application, usually linking together C++ and UPC object files, or calling a communication library. Either alternative defeats static type checking because a C++ compiler does not understand UPC types and general-purpose libraries deal with raw bytes. The solution is to develop a PGAS model for C++. As others have demonstrated, Fortran's coarray model can be ported directly to C++ using templates; however, this paper shows that a direct port is insufficient because it does not permit static type checking and C++ idioms. Instead, this paper presents the design of a new approach where these problems are addressed. The implementation is released as Coarray C++ in version 8.2 of the Cray Compiling Environment.

1 Introduction

Fortran [1] is the archetypal HPC programming language and C [3] is common, but increasingly often developers consider C++ [2] for their implementation language. Modern compilers make its performance competitive, many universities teach programming using C++, and its object-oriented features help to organize large applications. Similarly, the Message Passing Interface (MPI) library [12, 13] is the typical HPC parallel programming model, but PGAS languages like Fortran and UPC [17] offer an alternative that is often simpler due to one-sided communication requiring less coordination by the programmer. Although one-sided MPI [13] can provide that same feature, language-based approaches allow the compiler to help the programmer via static type checking.

Unfortunately for HPC application developers, no solid option has arisen to combine these trends to allow programming in C++ with a PGAS model. Currently C++ programmers have two options: write a mixed-language application or use a one-sided communication library. An example of the first option is to write most of the application in C++, but keep all communication and shared data in UPC source files. These parts are glued together with a C language interface that uses type punning because C and C++ compilers understand neither UPC `shared` data types nor pointers to them. For example, the UPC code below provides a C interface for obtaining a `shared` struct from another UPC thread. The struct contains a UPC pointer-to-shared, so its type declaration is not syntactically valid on the C++ side of the interface. Worse, the C++ code cannot portably declare a different structure that has the same layout because UPC does not require that a pointer-to-shared have a particular size or alignment. For example, the C++ code below might allow the interface to work using certain compilers. Using pointers to `void` or treating a UPC data structure as an array of `char` are other techniques commonly seen to interface UPC and C++ code. Therefore, static type checking is lost at the interface. An example of the second option for writing C++ PGAS programs is

to write the entire application in C++ and call a library like SHMEM [6] or Global Arrays [14] for communication. Such libraries are general purpose and deal with raw bytes or a limited set of fundamental types.

```
1  /* UPC Code */
2  struct S1 {
3      ...  /* other C data members */
4      shared int* p;
5      ...  /* other C data members */
6  };
7
8  shared struct S1 data[THREADS];
9
10 void get_from_thread( struct S1* local, int thread ) {
11     *local = data[thread];
12 }
13
14 /* C++ Code */
15 struct S2 {
16     ...  /* other C data members */
17     long blob[2];  /* hopefully the same size and alignment as p */
18     ...  /* other C data members */
19 };
20
21 extern "C" { void get_from_thread( struct S2* local, int thread ); };
```

A PGAS model designed for C++ can provide type-checked communication in a way that feels natural to C++ programmers by supporting C++ idioms, but the details of such a model have been an open question without much investigation. Common features of C and C++ kindle speculation about a hypothetical UPC++ language, but even though UPC is a PGAS model for C, it does not follow that it is the most appropriate model for C++. C and C++ are two distinct languages that abide by different language standards [2, 3] and have their own programming idioms. Specifically, the preferred mechanism for introducing new features to C++ is via the template library and not via grammar modification. This strategy allows new features to be prototyped without compiler modification and made available for comment, often via Boost [5], before being considered for the standard. UPC modifies the C grammar and a UPC++ language would need to modify the C++ grammar to maintain consistency for programmers. Although a `shared<T>` template could implement UPC `shared` types in C++, there are many complications. Briefly, an array of `shared<T>` would not allocate the correct amount of memory per UPC thread, plus array elements are not self-aware of their position in the array – knowledge that a UPC compiler uses to determine data locality. Therefore, T would need to be the array type and properties like block size and rank of the `THREADS` dimension would need to be indicated via additional template parameters. These conflicting strategies – grammar modification versus templates – give UPC++ a difficult path to official adoption, compounded by UPC never having been adopted by standard C. Finally, UPC offers a wide variety of data distribution options that intentionally hide the location of a data access, such that an array element can be accessed without knowing which UPC thread owns it. This distribution flexibility and access transparency is sometimes useful, but experience shows that many data distributions are out-performed by a more locality-aware block distribution [4, 10, 11] that resembles coarrays. Fortran coarrays have been shown to compare well to MPI and out-perform UPC versions of the same application [7].

The coarray model adopted by Fortran [1] adds an additional dimension, called a codimension, to a normal scalar or array type. The codimension spans instances of a Single-Program Multiple-Data (SPMD) application, called images, such that the scalar or array on each im-

age becomes a coarray. Each image has immediate access via processor loads and stores to its own coarray, which resides in that image's local partition of the global address space. By explicitly specifying an image number in the cosubscript of the codimension, each image may access other images' coarrays. Fortran permits multiple cosubscripts to be mapped to an image number. Although originally a Fortran idea [15], coarrays have been implemented with C++ templates [8] and Python modules [16].

The C++ coarrays implemented by [8] prototyped an early incarnation of the Fortran coarray concept to avoid the cost of modifying a Fortran compiler. Fortran coarray syntax was literally moved into C++ instead of considering approaches that are more idiomatic for C++ and that permit a greater degree of static type checking. Furthermore, C++ has changed since then. C++11 [2] introduced a shared-memory parallel programming model using threads. Coarray images are a broader and typically orthogonal concept to threads, representing cooperating processes in a distributed system where a given image might consist of multiple threads acting within a shared-memory domain. For consistency and ease of parallel programming, it makes sense that idioms developed for C++11 threads should influence how coarrays are implemented in C++. This paper presents the design and implementation of Coarray C++ in version 8.2 of the Cray Compiling Environment, where type safety and maintaining the "look and feel" of standard C++ are of paramount concern. Implementing Coarray C++ did not require modifying the compiler or runtime libraries, which is typical for C++ template-based programming models [8, 9].

2 Coarray C++ "Hello World"

The following program is the Coarray C++ equivalent of the classic "Hello World" program:

```
1 #include <iostream>
2 #include <coarray_cpp.h>
3 using namespace coarray_cpp;
4 int main( int argc, char* argv[] ) {
5     std::cout << "Hello from image " << this_image()
6               << " of " << num_images() << std::endl;
7     return 0;
8 }
```

The header file coarray_cpp.h included by Line 2 provides all Coarray C++ declarations within `namespace coarray_cpp`. Normally a program imports all of the declarations into its namespace with a `using` directive as on Line 3, but having the `coarray_cpp` namespace grants the programmer the flexibility to deal with name conflicts. The functions `this_image()` and `num_images()` called on Lines 5 and 6 return the current image's zero-based rank and the total number of images in the job. Note that in [8], these functions curiously were member functions, such that one needed a coarray object just to find out image information. The `this_image()` function is vital for the SPMD practice of branching on the image number; global scope is important for cases where no coarray is in scope. The program is compiled with the Cray compiler and executed using four images as follows:

```
> CC -o hello hello.cpp
> aprun -n4 ./hello
Hello from image 0 of 4
Hello from image 1 of 4
Hello from image 2 of 4
Hello from image 3 of 4
```

The Cray compiler automatically links the application with the same PGAS language runtime library used for Cray UPC and Cray Fortran, as well as a networking library. It is possible to use other C++ compilers, such as the GNU g++ compiler, on a Cray system to build Coarray C++ applications, but the user must link with the necessary libraries explicitly. For Coarray C++ implemented for a different platform, the types and function signatures within coarray_cpp.h would remain the same, but their implementation, the runtime libraries, and the mechanism for launching the application would be different.

3 Type System

3.1 Coarrays

Coarray C++ has a generic coarray template with several specializations. Their forward declarations are shown below:

```
1 template < typename T >           class coarray;
2 template < typename T >           class coarray<T[]>;
3 template < typename T, size_t S > class coarray<T[S]>;
4 template < typename T >           class coarray< coatomic<T> >;
```

Line 1 is the generic coarray template, Line 2 is specialized for unbounded arrays (an array where the leftmost extent is unspecified at compile time), Line 3 is specialized for bounded arrays, and Line 4 is specialized for coatomic, which serves the same purpose for images as the `std::atomic<T>` template does for C++11 threads (see Section 5.2). These specializations let the coarray template provide type-appropriate constructors, member functions, and operators. Coarrays of pointers are handled sufficiently by the generic template, but have special properties (see Section 3.6). For bounded array types, all array extents appear as part of the template argument; for an unbounded array type, the leftmost extent is specified at runtime via a constructor argument:

```
1 coarray<int> i;
2 coarray<int[10][20]> x;
3 coarray<int[][20]> y(n);
```

An important distinction from [8], where the declaration for x at Line 2 would be `CoArray<int> x(10, 20)`, is that the extents are part of the type of the coarray so that the compiler can help enforce type safety. Note that for unbounded array types, the compiler has partial extent information. The C++ type system permits only the first array extent to be unbounded, thus in Coarray C++ one cannot declare a coarray where multiple extents are specified at runtime. This restriction matches C++ because it does not permit allocating multidimensional arrays where a non-leading dimension is variable (e.g., `new int[10][n]` is a compile-time error whereas `new int[n][20]` is fine).

A coarray declaration creates a coarray object which allocates and manages an object of its template argument type in the current image's partition of the global address space. Both the creation and destruction of a coarray must be executed collectively by all images. This requirement is satisfied automatically for global and static local coarray declarations, but the programmer is responsible for ensuring it for local and dynamically-allocated coarrays. Although image teams would allow this requirement to be waived, Coarray C++ does not yet have teams. The Fortran standard committee is considering how to implement teams and following their lead might make sense for Coarray C++.

The Cray implementation allocates the managed objects at symmetric virtual addresses across all images, so an all-to-all communication of addresses during construction, as done by [8], is unnecessary. A scalar type is assumed to have a default constructor; if it also has a copy constructor, then the `coarray<T>` constructor accepts an initializer of type T which may have a different value on different images. For an array type, the ultimate element type of the array is assumed to have a default constructor and initializers are not supported. Internally, these constructors are called using the C++ language's placement `new` syntax, passing the symmetric memory address.

3.2 Local Data Access

A `coarray<T>` object transparently behaves like the local T object that it manages, in both l-value and r-value contexts, so that an existing variable declaration of type T can be changed to a coarray by modifying its declaration without having to modify all uses of the variable:

```
1 i = 0;
2 x[1][2] = i;
3 y[3][4] = x[1][2];
```

Assuming the same declarations as in Section 3.1, `i`, `x[1][2]`, and `y[3][4]` all act as a reference to an object of type `int` on the current image. The assignments compile to processor loads and stores and permit normal compiler optimizations like forward substitution. Local assignments within loops may be vectorized for targets that support vectorization.

Note that in [8], the access `x[1][2]` would need to be changed to `x(1, 2)`, which is strange for C++ programmers (though familiar to Fortran programmers). Beyond seeming strange, the different syntax prevents the C++ practice of writing generic code, such as a function template that accepts either a normal array or a coarray. Member access is the only exception to local-access syntax transparency due to C++ operator overloading limitations. C++ does not permit the member access (dot) operator to be overloaded, so accessing a member of a coarray of struct type requires switching to the arrow operator:

```
1 struct Point { int x, y; };
2 coarray<Point> pt;
3 pt->x = 1;
4 pt->y = pt->x;
```

This change is familiar to any C or C++ programmer who has changed an existing variable declaration to pointer type.

3.3 Coreferences

Coreferences extend the C++ concept of references to potentially remote objects that reside in coarrays. Similar to the coarray template, there is a generic `coref` template and multiple specializations to provide type-specific behavior. These types are discussed in Sections 5.2, 6.2, and 6.3.

```
1 template < typename T >           class coref;
2 template < typename T >           class coref<T[]>;
3 template < typename T, size_t S > class coref<T[S]>;
4 template < typename T >           class coref<T*>;
5 template < >                      class coref<coevent>;
6 template < >                      class coref<comutex>;
7 template < typename T >           class coref< coatomic<T> >;
```

To access data on other images, the image number is placed in parenthesis immediately after the coarray object. Any square brackets for array subscripts follow these parenthesis, so `x(5)[1][2]` is `x[1][2]` on image 5. In [8], the syntax would be `x[5](1, 2)`, which uses the Fortran syntax of square brackets for the image number. Having already established in Section 3.2 that square brackets must continue in their familiar role for C++ array element access, it makes more sense to use parenthesis for the cosubscript to maintain a visual distinction for subscripts and cosubscripts. The following assignments compile to code capable of one-sided gets and puts across a network:

```
1 i(7) = 0;                  // put
2 x[1][2] = i(6);            // get
3 y(0)[3][4] = x(5)[1][2]; // get then put
```

From a type perspective, `x(5)[1][2]` is a call to `operator()` of coarray x, which returns a coreference of type `coref<int[10][20]>`. The `[1]` bracket calls `operator[]` of `coref<int[10][20]>`, which returns a `coref<int[20]>`. Finally the `[2]` bracket calls `operator[]` of `coref<int[20]>`, which returns a `coref<int>`. This call sequence is inlined. The `coref<int>` behaves as an `int` in l-value and r-value expression contexts, except that reading it will get data from image 5 and writing it will put data to image 5. If x had been `const` in the context of the access, a `const_coref` would be obtained from the `operator()` call instead. A `const_coref` does not permit modification of data. Accessing members on other images is again complicated by C++ limitations – a pointer to the member must be used:

```
1 pt(5).member( &Point::x ) = 1;
```

The above code sets `pt.x` equal to 1 on image 5. Overloading the `->*` operator was investigated, but it has a different precedence than the dot and arrow operators, which caused confusion in some contexts. Calling member functions of objects on other images is not supported.

3.4 Traits

The assignments in Section 3.3 made bitwise copies of fundamental data types, but C++ objects may have non-trivial copy semantics. C++11 has a template, `std::is_pod<T>`, to check if a type is a "plain old data" type that does not require special semantics; however, Coarray C++ does not yet require a C++11 compiler and `std::is_pod<T>` cannot be usefully emulated without compiler support. To solve this problem, Coarray C++ has the following template:

```
1 template < typename T >
2 struct coarray_traits {
3 static const bool is_trivially_gettable = true;
4 static const bool is_trivially_puttable = true;
5 };
```

Various `coref<T>` member functions consult `coarray_traits<T>` to determine if copying an object's bits is sufficient. A user can specialize the template for a non-trivial type of their own creation. Indicating that a type is not trivially gettable enters a contract for the type to have a remote copy constructor and a remote assignment operator that accept a `const_coref<T>`. These functions can use the `const_coref<T>` to read enough information to calculate how much local storage is required to copy the object, allocate sufficient space, then copy the rest of the remote object's data. Indicating that a type is not trivially puttable prohibits the type from being written in a one-sided manner to another image because it would require help from the target image.

3.5 Copointers

Just as any C++ object can have its address taken, a `coref<T>` has an `address()` member function that returns a `coptr<T>`. Likewise, a `const_coptr<T>` is obtained from a `const_coref<T>`. The `&` operator was not overloaded in case the programmer wants a pointer to a coreference, but one can write a standalone `&` operator that calls `address()`. Copointers support pointer arithmetic and can be used as iterators with standard C++ algorithms. Unlike coreferences, they can be null. Unlike (most) UPC pointers, arithmetic on them never changes the target image. The following code fills the array on image 2 with the value 42:

```
1 #include <algorithm>
2 coarray<int[100]> x;
3 coptr<int> begin = x(2)[0].address();
4 coptr<int> end = x(2)[100].address();
5 std::fill( begin, end, 42 );
```

Local pointers are convertible to copointers; going the other direction, copointers have a `to_local()` member function that attempts to return a normal C++ pointer to the same data. If the copointer targets the current image, then this function will succeed. If the copointer targets a different image within the same shared-memory domain, then the function may be able to return a special address that is mapped to the original object. In all other cases, `to_local()` returns NULL.

3.6 Coarrays of Pointers

A coarray allocates the same amount of memory on every image, but this approach sometimes can waste memory. A coarray of pointers mirrors a Fortran feature where one can create a coarray of a derived type containing a pointer component that is then associated with a different amount of memory on each image. Coarray C++ uses the specialization `coref<T*>` to provide the necessary behavior:

```
1  coarray<int*> x;
2  x = new int[this_image() * 10];
3  *x = this_image();
4  x[4] = this_image();
5  sync_all();
6  ...
7  int y = *x(2);
8  y += x(3)[4];
9  ...
10 sync_all();
11 delete [] x;
```

In the above code, x acts like an `int*` and can hold the result of an allocation via `new` on Line 2. Lines 3 and 4 show that x can be dereferenced locally using normal syntax. The `sync_all()` calls (see Section 6.1) on Lines 5 and 10 ensure that other images do not read the current image's data before it is allocated and initialized or after it is deallocated. The remote accesses on Lines 7 and 8 work because the cosubscript returns a `coref<int*>`, which first reads the pointer from the target image before issuing a second read to the pointer's target *on the target image*. For repeated access to the same data, such as within a loop, this extra read can be hoisted manually via a copointer:

```
1 const_coptr<int> p = x(3)[0].address();
2 for ( int i = 0; i < 30; ++i )
3     foo( p[i] );
```

3.7 Coarrays and Functions

A coarray may be passed to a function via a reference or a pointer, but may not be passed by value. If a coarray could be passed by value, the call would have to be collective. There would be a collective allocation of a temporary coarray, the data within the original coarray would need to be copied into the temporary coarray, and eventually the temporary coarray would need to be collectively destroyed. Pass by value is expensive and there are better alternatives, like passing a coarray as a const reference, so it is a compile-time error.

4 Type Checking

4.1 Static Checking

Coarray C++ types whose shapes are completely known at compile time are statically type checked by the C++ compiler. The following example shows a type error detectable by the compiler:

```
1 void foo( coarray<int[10][20]>& x );
2 coarray<int[5][10]> y;
3 foo( y ); // illegal
```

A bounded type is convertible to an unbounded type:

```
1 void foo( coarray<int[][20]>& x );
2 coarray<int[10][20]> y;
3 foo( y ); // legal
```

4.2 Dynamic Checking

An unbounded type is convertible to a bounded type, but may throw a `mismatched_extent_error`:

```
1 void foo( coarray<int[10][20]>& x );
2 coarray<int[][20]> y(n);
3 foo( y ); // throws if n != 10
```

4.3 shape_cast

For instances where a coarray or coreference of one shape needs to be reinterpreted as a different shape, `shape_cast` provides a runtime conversion. The conversion works provided that the ultimate element type matches and the new type does not have more elements than the old type, otherwise it throws `std::bad_cast`. The syntax is modeled after the other C++ casts: `static_cast`, `dynamic_cast`, `const_cast`, and `reinterpret_cast`. None of those are sufficient to provide the same functionality as `shape_cast`.

```
1 void foo( coarray<int[10][20]>& x );
2 coarray<int[200]> y;
3 coarray<int[50]> z;
4 foo( shape_cast<int[10][20]>(y) ); // legal
5 foo( shape_cast<int[10][20]>(z) ); // throws
```

5 Memory Model

5.1 atomic_image_fence

Coarray C++ follows the host compiler's C++ memory model for local accesses and the Fortran model for accesses to other images. Accesses by a single image appear to execute in program order. Prior to executing an `atomic_image_fence()` call, which is modeled after C++11's `atomic_thread_fence()`, a write to another image need only be visible to the image that wrote the value. After the fence, the write is visible to all images. Therefore, an implementation of Coarray C++ is free to use non-blocking communication for all writes, provided that it can ensure program order when the same image makes multiple accesses to the same data. Reads block so that a coarray used in an expression context can provide a value. Therefore, coreferences provide a `get()` member function to launch a non-blocking read that is not guaranteed to complete until the next fence.

5.2 coatomics

C++11 introduced the `atomic<T>` template for constructing atomic types. All operations on these types are atomic with respect to C++11 threads. Likewise, Coarray C++ has `coatomic<T>` to provide operations that are atomic with respect to images. Similar convenience typedefs are provided, like `coatomic_long` for `coatomic<long>`. A generic coatomic type uses a comutex to provide atomicity (see Section 6.3), but implementations of Coarray C++ may specialize certain types, like `coatomic<long>`, to use lock-free hardware atomics. In distributed systems, `atomic<T>` operations would use processor atomics whereas `coatomic<T>` operations would use network atomics; these two sets of atomic operations might not be memory coherent and atomic with respect to each other. The following code shows an atomic addition:

```
1 coarray<coatomic_long> x(0L);
2 size_t n = num_images();
3 for ( size_t i = 0; i < n; ++i ) {
4     x(i) += this_image(); // atomic add
5 }
6 sync_all();
7 assert( x == ( n * ( n - 1 ) / 2 ) );
```

Atomic operations may be performed on regular types by explicitly creating a coreference to the matching atomic type, but it is left up to the programmer to ensure that non-atomic operations do not simultaneously touch the data.

```
1 coarray<long> x(0L);
2 coref<coatomic_long> ref( x(i) );
3 ref += this_image(); // atomic add
```

6 Image Synchronization

6.1 sync_all

As with Fortran's `sync all` statement, calling `sync_all()` synchronizes control-flow across all images and implies a fence. A direct equivalent to Fortran's `sync images` statement for point-to-point image synchronization is not provided, but see Section 6.2 below. Experience shows that programmers use `sync images` in contexts where `sync all` is more appropriate and expect equivalent performance, which is not realistic.

6.2 coevent

Events are under consideration for inclusion in the Fortran standard. Coevents provide similar behavior, allowing one image to post an event and another image to wait on an event to be posted. A `post()` call is directed at a particular image, but the `wait()` call does not know which image posted the event:

```
1 coarray<coevent> events;
2 if ( this_image() == 0 ) {
3     // write something to image 1, then
4     events(1).post();
5 }
6 else if ( this_image() == 1 ) {
7     events->wait();
8     // then read the data
9 }
```

The specialization `coref<coevent>` provides the `post()` function in the above example. Waiting on non-local events (e.g., `events(1).wait()`) is not supported.

6.3 comutex

A comutex is modeled after C++11's `std::mutex`. A comutex provides mutual exclusion among images. It is up to the programmer to establish a relationship between a comutex and the data that it protects, although most sensible programming styles dictate that acquiring a mutex on an image implies that the mutex guards data on that image:

```
1 coarray<comutex> m;
2 m(i).lock();
3 // access data on image i
4 m(i).unlock();
```

7 Cofutures

The facility for explicit management of non-blocking communication is based on C++11's `std::future<T>` template. In C++11, a `std::future<T>` manages completion of an asynchronous operation that produces a T value. Likewise, in Coarray C++, a `cofuture<T>` manages a non-blocking copy of type T. A `coref<T>` can provide a `cofuture<T>`, either by calling `get_cofuture()` or relying on implicit conversion:

```
1 coarray<int> x;
2 ...
3 cofuture<int> f = x(i);
4 ...
5 int z = f + 1;
```

Using the cofuture in an expression context automatically waits on the data to arrive. If the data is large, existing storage may be preferable to duplicating storage inside a cofuture. In that case, T is `void` because the cofuture does not store data and the `wait()` member function or the destructor ensures completion:

```
1 coarray<int[100]> x;
2 int y[100];
3 ...
```

```
4 cofuture<void> f = x(i).get_cofuture(&y);
5 ...
6 f.wait();
```

Finally, a non-blocking write can be explicitly managed via `put_cofuture()`:

```
1 coarray<int> x;
2 int y;
3 ...
4 cofuture<void> f = x(i).put_cofuture(&y);
5 ...
6 f.wait();
```

8 Collectives

8.1 cobroadcast

Cobroadcast replicates the value of a coarray from a root image across all images. The broadcast does not imply a `sync_all()` because synchronization is not needed when only local values are accessed, as in this example:

```
1 coarray<int> x;
2 if ( this_image() == 0 ) {
3     x = 42;
4 }
5 cobroadcast( x, 0 );
6 assert( x == 42 );
```

8.2 coreduce

Coreduce performs a broadside reduction of coarray images. For example, reducing a `coarray<int[100]>` yields 100 result values instead of one. Like cobroadcast, no `sync_all()` is implied. By default, every image receives the results as part of the original coarray, but there are options to send the result to only one image or to use a different coarray for the results. Coreduce accepts a commutative and associative function, but implementations may provide optimized specializations with alternative names. For example:

```
1 coarray<int[100]> x;
2 cosum( x ); // coreduce( x, std::plus<int> )
3 comin( x ); // coreduce( x, std::less<int> )
4 comax( x ); // coreduce( x, std::greater<int> )
```

An example of using comax:

```
1 coarray<int> x;
2 x = this_image();
3 comax( x );
4 assert( x == num_images() - 1 );
```

9 Performance Considerations

Because C++ compilers are not aware of Coarray C++, performance would appear to be a huge challenge. This misconception stems from a belief that compilers for PGAS languages only

achieve high-performance by exploiting the language's memory consistency model to schedule and cache remote accesses. Although UPC, and to some extent Fortran, were designed with these optimizations in mind, most of the performance of Cray UPC and Fortran comes from other techniques. The primary network latency optimization is to issue non-blocking writes for all writes, relying on the language runtime to efficiently enforce the memory model. Coarray C++ behaves identically. The primary network throughput optimization is to recognize loops that do small, constant-stride remote memory accesses and replace them with bulk copy functions. UPC has a family of functions (e.g., `upc_memput()`, `upc_memget()`) for this purpose; both UPC loops and Fortran array syntax loops can be mapped to similar functions. To provide this behavior in C++, which does not permit array syntax, Coarray C++ allows a coreference to an array to act as the source or target of an assignment. For syntactic convenience, a `make_coref()` function automatically creates a `coref` from a local object:

```
1 coarray<int[10][100]> x;
2 int local[10][100];
3 ...
4 make_coref( local ) = x(2);
5 x(7)[3] = local[3];
```

Line 4 creates a `coref<int[10][100]>` from *local*, then uses it to store the contents of x from image 2. There also is support for copying complete slices of arrays. Line 5 copies just row 3 to image 7. Experiments with these techniques have shown performance equivalent to Cray UPC and Fortran.

10 Conclusions

Coarray C++ provides typical PGAS language features in a manner that is compatible with C++ idioms and that permits static and dynamic type checking. Throughout this paper, its design was contrasted with a more literal approach [8] to moving Fortran features into C++ that did not have the benefit of building on C++11 parallel programming ideas. The initial version of Coarray C++ is released with version 8.2 of the Cray Compiling Environment and uses the same language runtime and networking libraries as Cray UPC and Fortran. Future work is to evolve the language based on user experience, as well as any interesting developments in both the Fortran and C++ standards. Implementations for other platforms are encouraged.

Acknowledgements

The author thanks David Henty for presenting this paper, Bill Long for insight into Fortran's coarray rationale, Steve Vormwald for discussions comparing UPC and coarrays, Kristyn Maschhoff for taking the time to report bugs while undertaking a large project written in Coarray C++, and Harvey Richardson and the anonymous reviewers for their helpful comments.

References

[1] Fortran Standard: ISO/IEC 1539-1:2010. 2010.

[2] C++ Standard: ISO/IEC 14882:2011. 2011.

[3] C Standard: ISO/IEC 9899:2011. 2011.

[4] K. Berlin, J. Huan, M. Jacob, G. Kochhar, J. Prins, B. Pugh, P. Sadayappan, J. Spacco, and C. wen Tseng. Evaluating the Impact of Programming Language Features on the Performance of

Parallel Applications on Cluster Architectures. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pages 194–208.

[5] Boost C++ Libraries. www.boost.org.

[6] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Models*, 2010.

[7] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An Evaluation of Global Address Space Languages: Co-array Fortran and Unified Parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 36–47, 2005.

[8] M. Eleftheriou, S. Chatterjee, and J. E. Moreira. A C++ Implementation of the Co-Array Programming Model for Blue Gene/L. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)*, 2002.

[9] M. Garland, M. Kudlur, and Y. Zheng. Designing a Unified Programming Model for Heterogeneous Machines. In *SC12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[10] N. Jansson. Optimizing Sparse Matrix Assembly in Finite Element Solvers with One-Sided Communication. *VECPAR 2012, Lecture Notes in Computer Science*, 7851:128–139, 2013.

[11] H. Jin, R. Hood, and P. Mehrotra. A Practical Study of UPC with the NAS Parallel Benchmarks. In *Proceedings of the Third Conference on Partitioned Global Address Space Programming Models*, 2009.

[12] MPI Forum. MPI: A Message-Passing Interface Standard, Version 2.2. 2009.

[13] MPI Forum. MPI: A Message-Passing Interface Standard, Version 3.0. 2012.

[14] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[15] R. W. Numrich. A Parallel Extension to Cray Fortran. *Scientific Programming*, 6:275–284, 1997.

[16] C. E. Rasmussen, M. J. Sottile, J. Nieplocha, R. W. Numrich, and E. Jones. Co-array Python: A Parallel Extension to the Python Language. *Euro-Par 2004, Lecture Notes in Computer Science*, 3149:632–637, 2004.

[17] UPC Consortium. UPC Language Specification 1.3 (Draft). September 2013.

Introducing SHMEM into the GROMACS molecular dynamics application: experience and results

Ruyman Reyes[1], Andrew Turner[1] and Berk Hess[2]

[1] EPCC, University of Edinburgh, Edinburgh, UK
rreyesc,aturner@epcc.ed.ac.uk
[2] Berk Hess, KTH, Stockholm, Sweden
hess@kth.se

Abstract

In order to take advantage of exascale-level computing, it is necessary to extract performance from large number of cores distributed in many nodes. To accomplish this, new programming models need to be explored and techniques to port codes to these new programming models have to be developed. GROMACs, a leading molecular dynamics open-source package, is one of this codes whose scalability needs to be analysed and improved for exascale machines given the interest in the scientific community. Although it has been already ported to hybrid MPI and OpenMP, there is still room for improvement in situations involving a large number of nodes. In this work, we present the experiences of porting GROMACs, a molecular dynamics package, to SHMEM, a low-latency, one-side communication library. Results show that the replacement of MPI routines by one-side equivalents does not affect performance, although current limitations on the programming model and the implementation limits the performance benefit.

1 Introduction

The open-source GROMACS[5]simulation package is one of the leading biochemical molecular simulation packages in the world and is widely used for simulating a range of biochemical systems and biopolymers. This is reflected in the fact that both the PRACE pan-European HPC initiative and the CRESTA exascale collaborative project have identified GROMACS as a key code for effective exploitation of both current and future HPC systems [1].

The GROMACS code has recently undergone a major restructuring to be able to take advantage of both hybrid MPI/OpenMP programming models and to be able to exploit GPU accelerators. A key consideration of a hybrid programming model such as this is the balance that needs to be achieved between the number of OpenMP threads and MPI tasks. The performance of the MPI communications have a large influence on both the scaling of the code and the number of OpenMP threads per task that can be exploited effectively. If the performance of the communications between parallel tasks could be improved then this would lead to being able to run the code with a lower number of particles per parallel task improving both the scaling and parallel efficiency of GROMACS.

In this work we aimed to improve the performance of the inter-task communication by replacing the calls to standard MPI two-sided communication routines with a single-sided communication interface which can be implemented using different single-sided communication libraries (for example, SHMEM, MPI 3, Fujitsu Tofu interconnect, Infiniband verbs). Our development has been performed on the HECToR UK National Supercomputing Facility[3]. HECToR is a Cray XE6 machine and so we have implemented the interface using calls to the single-sided,

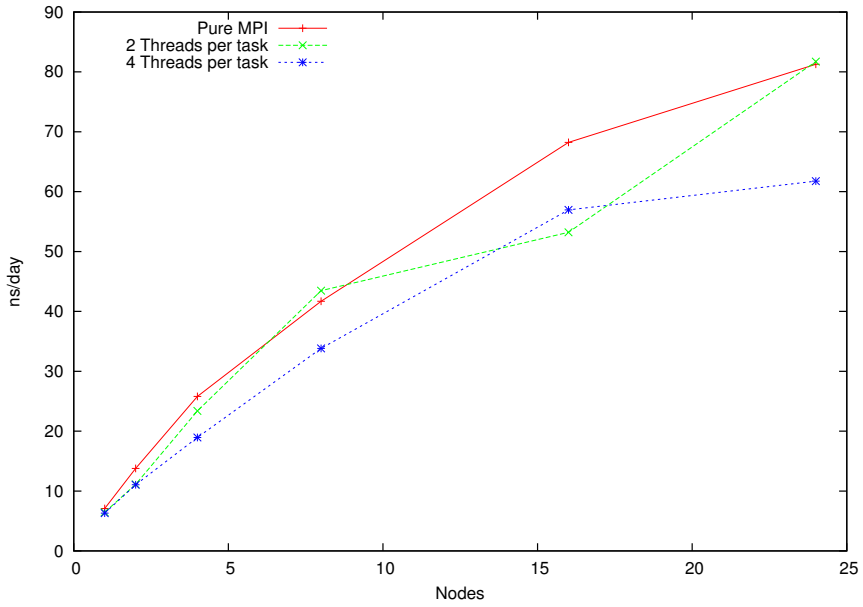


Figure 1: Performance of GROMACS on HECToR Phase 3 for ADH Cubic test case.

Cray SHMEM communication routines. SHMEM has been selected as it currently available in a form that can deliver excellent performance on Cray MPP machines (such as HECToR) and is also currently an area of active development through the OpenSHMEM[2] initiative.

1.1 Initial Benchmarking and Profiling

1.1.1 GROMACS 4.6

Our initial benchmarking and profiling are based on the ADH Cubic test case supplied by the GROMACS developers using both pure MPI and hybrid MPI+OpenMP (with multiple OpenMP threads per MPI task). The scaling as a function of number of HECToR nodes is illustrated in Figure 1.

The data shows that the pure MPI version of GROMACS scales well up to 24 nodes (768 tasks) for this benchmark. The hybrid version with two OpenMP threads per MPI task generally shows lower absolute performance but is able to exploit more cores (up to 32 nodes, 1024 tasks). Using higher numbers of MPI tasks for this benchmark is problematic due to issues matching the parallel domain decomposition to the larger task count.

Table 1.1.1 provides an overview of where time in the code is spent when using both 64 and 512 MPI tasks. We observe that, as expected, when we increase the number of tasks, the MPI communications become a significant portion of the calculation. In particular, the MPI point-to-point routines (MPI_Sendrecv and MPI_Recv) take up the majority of the communication time. Profiling also reveals that as the number of tasks is increased the sizes of the messages become smaller - at 24 nodes more than 80% of the MPI_Sendrecv messages are less than 4kB in size.

Class	Function	24 MPI Tasks	512 MPI tasks
USER		76.7%	49.1%
	Main	76.7%	49.1%
MPI		22.5%	47.4%
	MPI_Recv	4.6%	27.1%
	MPI_Sendrecv	6.7%	11.3%
	MPI_Alltoall	1.2%	5.0%
	MPI_Waitall	9.6%	3.5%

Table 1: Percentage of time spent in MPI communication routines of the initial profiled version.

1.1.2 SHMEM

The SHMEM library is an implementation of a single-sided communications library where data is transferred between parallel tasks by copying or reading data directly from remote memory. This contrasts with the traditional, two-sided MPI approach where the two tasks must handshake to exchange data. We would expect the single-sided approach to be faster as it involves fewer software layers and memory copy operations per communication call.

The SHMEM approach should be particularly efficient on the Cray XE architecture as the Gemini interconnect has dedicated hardware support for single-sided communications via Remote Direct Memory Access (RDMA) using the low-level DMAPP API. There are very few layers of software between the SHMEM interface and the Gemini hardware as each SHMEM call generally maps directly onto a low-level DMAPP call [6].

We have tested this by implementing a SHMEM equivalent of the MPI_Sendrecv subroutine and then compared the performance between the two as a function of message size on the current HECToR Phase 3 hardware. The results show that for small message sizes ($<$ 2KB) the SHMEM operation is around 3 times faster than the MPI equivalent and for larger messages ($>$4KB) the SHMEM operation are around 30% faster.

2 Implementation

We have started with replacing the MPI point-to-point communications in the domain decomposition part of the code. Converting an MPI code to SHMEM is conceptually easy. Listing 1 illustrates a classical MPI_SendRecv operation where each processor sends data to the one on its left and receives from the one on its right.

Listing 1: Simple Send Receive code in MPI

```
{
int right = (my_rank + 1) % numprocs;
int left = my_rank - 1;
left = left<0?numprocs-1:left;

MPI_SendRecv(send_data, nelem, MPI_BYTE, left, tag,
             recv_data, nelem, MPI_BYTE, right, tag,
             MPI_COMM_WORLD, &status);
}
```

The SHMEM equivalent of this operation is shown in Listing 2. Notice that in this case, although for the sake of clarity we keep computing right and left, we only need left. All ranks will put data on the left rank, thus, implicitly they are receiving data from the right rank. A

similar implementation, using get instead of put would get the data from the rank in the right. To ensure all ranks have receive the data we use a global barrier.

Listing 2: Simple Send Receive code in SHMEM

```
{
int right = (my_rank + 1) % numprocs;
int left = my_rank - 1;
left = left<0?numprocs-1:left;

shmem_putmem(&recv_data, send_data, nelem*sizeof(recv_data[0]), left);
shmem_barrier_all();
}
```

Although at a first glance the replacement may be seen straightforward, in the example we make some assumptions that simplify the implementation. Note that four restrictions should be considered when porting larger code bases (as in the case of GROMACS):

1. SHMEM is a SPMD model
2. The data to be put or to be get from needs to reside on symmetric memory,
3. Symmetric heap memory routines force an implicit global synchronization,
4. One-side operations are asynchronous by nature,

We will describe how we overcame aforementioned restrictions in our GROMACS SHMEM implementation.

2.1 SHMEM SPMD limitations

A notable limitation of the SHMEM programming model is that it follows the Single Program Multiple Data model (SPMD): No process can be added or removed from the group and all processes execute the same application. This contrast with the GROMACS Multiple Program Multiple Data (MPMD) approach to solve the Particle-Mesh-Ewald method (PME) on a predefined subset of nodes. To facilitate the work we have disabled this feature in the command line, assuming all nodes perform the PME part of the code. It is possible to overcome this limitation with additional work in the non-PME part of the code but we could not implement it within the time allocated to this project.

2.2 Porting memory allocations to the Symmetric Heap

In Listing 1 we assume that the data pointed by `recv_data` resides on the symmetric heap. In C, non-stack variables (such as global or static variables) are symmetric across all ranks (PEs in SHMEM nomenclature), which excludes memory allocated using traditional *malloc*. As an example, the current GROMACS MPI implementation makes extensive use of temporary buffers allocated using *realloc* in each rank. This temporary buffers would be freed later, and are adjusted to the size of the data being sent or received. Restriction (1) forces us to use a symmetric buffer for these communications. However, if not all ranks reallocate at the same point, the implicit global barrier will cause a deadlock. The code in GROMACS depicted in Listing 3 is used in several places across the code to minimize the memory footprint. The local buffers are reallocated before a communication take place if the data to be received is bigger than the current size of the buffer. In the SHMEM implementation, it is necessary to ensure that the size of the buffer is the same across all ranks, hence forcing a global max computation

before the reallocation (Listing 4). To avoid over-synchronization, instead of computing the size before the reallocation itself we added an additional parameter to the GROMACS data structures communicated (mainly atoms and forces) storing the maximum number of elements in all PEs, reducing the number of calls to the global maximum collective.

Listing 3: Simple buffer reallocation in the MPI implementation

```
if ( local_buffer_size > number_of_elements * size )
{
    local_buffer_size = number_of_elements * size;
    buffer = realloc(buffer, local_buffer_size);
}
```

Listing 4: Buffer reallocation when using SHMEM

```
int max_local = shmem_get_max_alloc(shmem, local_buffer_size);
if ( max_local >= number_of_elements * size )
{
    max_local = number_of_elements * size;
    buffer = sh_srealloc(buffer, max_local);
}
```

To reduce the cost of computing the maximum size across ranks, we can take advantage of the nature of the molecular dynamics simulation. After a certain number of iterations, the communication buffers stop growing. We have implemented a simple heuristic where if after a certain number of iterations the buffer no longer grows, the reallocation is disabled. Currently, the number of iteration is set by the user using a compile-time variable, but this could easily be set using an environment variable.

2.3 PE synchronization

The simplest solution to deal with restriction (3) is to use a global barrier to ensure that all processors have received the data, as shown before in Listing 2. Although given the nature of GROMACS communication pattern this is possible, it is not reasonable from the point of view of performance.

To ensure that the receiver has the data the sender is putting, a flag can be used to monitor the status, so that the receiver can wait until the data has been put in the buffer buy the sender.

In addition to this point-to-point synchronization, it is necessary to avoid race-conditions when the SHMEM communications are enclosed in a subroutine. Lets take for example the routine depicted in Listing 5. If the routine is called only once, or if it is always called to communicate the same pair of ranks, the routine will work as expected. However, if the routine is called consecutively with different pairs of ranks to perform the send/recv operation, the value of the flag may be overwritten, breaking the synchronization.

A lock alone would not solve the problem, as SHMEM locks are served on a first-come first-served policy. If a rank runs overtakes another, it may acquire the lock even if it is not its turn. To avoid this problem, and taking advantage of the fact that SendRecv communications in GROMACs are performed by all PEs, we use a static counter in each routine that its incremented with each call. When a PE enters the routine, it gets the remote value of the counter. If the number matches its own, then it continues. If not it just waits in a loop until the other PE reaches the same value for the calling counter. Using this mechanism we avoid global barriers and we can reduce the number of idle PEs. A diagram of the Send Receive routine is shown in Figure 2.

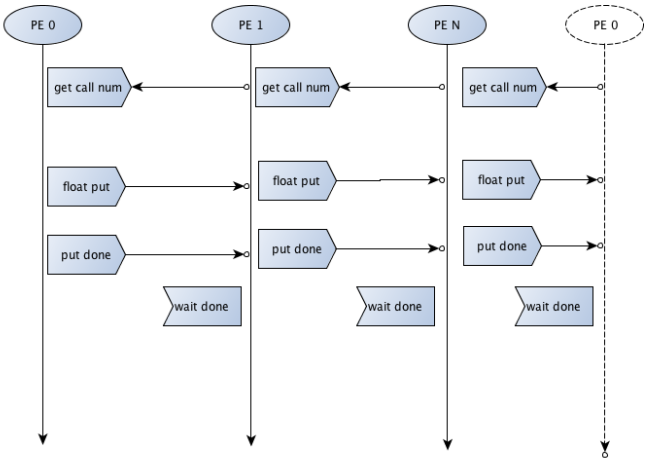


Figure 2: Diagram of a Send/Receive operation implemented on SHMEM

Listing 5: SHMEM Send Receive operation

```
void shmem_sendrecv_float(....)
{
   static int flag = -1;
   static int counter = 0;

   counter++;
   wait_for_previous_call(&counter)

   if (data_to_send)
   {
      shmem_float_put(recv_buf, send_buf, ..., dest_rank);
   }
   shmem_quiet();
   shmem_int_p(&flag, 1, dest_rank);

   shmem_int_wait(&flag, -1);
   flag = -1;
}
```

3 Performance results

3.1 Time spent in communication routines

Table 3.1 shows the output of the Craypat profiler tool when executing 100.000 steps of one of the performance benchmarks chosen. The last row of the table, Total comm, shows the sum of the time spent in MPI and the time spent in SHMEM. Notice the drastic reduction in the time spent in communication when using SHMEM. However, it is worth noting that the user time has increased. This is caused by the active waiting of the *shmem_wait_for_previous_iteration*. This time comes from the imbalance of the application and the problem being executed. The imbalance time in the MPI implementation is shown as time spent in SendRecv or Recv, but this comes to user code in our implementation. Notice however that the *shmem_wait* routines,

Group	1 node, 32 cores		16 nodes, 512 cores	
	MPI (s)	SHMEM (s)	MPI (s)	SHMEM (s)
Total	2231.71	2189	480.01	530.52
USER	1775.49 (79.6%)	1979 (90.4%)	144.00 (30.0%)	195.54 (37%)
MPI	411.31 (18.4%)	31.90 (1.5%)	257.38 (53.6%)	54.46 (10.3%)
MPI_SendRecv	310.99 (13.9%)	31.76 (35.5%)	109.46 (22.8%)	49.83 (9.4%)
MPI_AlltoAll	43.90 (2.0%)		126.25 (26.3%)	
MPI_Sync	44.90 (2.0%)	1.83 (0.1%)	78.62 (16.4%)	1.68 (0.3%)
SHMEM		61.53 (2.8%)		65.69 (12.4%)
shmem_wait		27.87 (1.3%)		11.16 (2.1%)
shmem_int_g		11.82 (0.5%)		22.71 (4.3%)
Total comm.	411.31 (18.4%)	93.43 (4.2%)	336 (70%)	121.83 (23%)

Table 2: Profiling output of an execution of the ADH testcase with 100.000 steps.

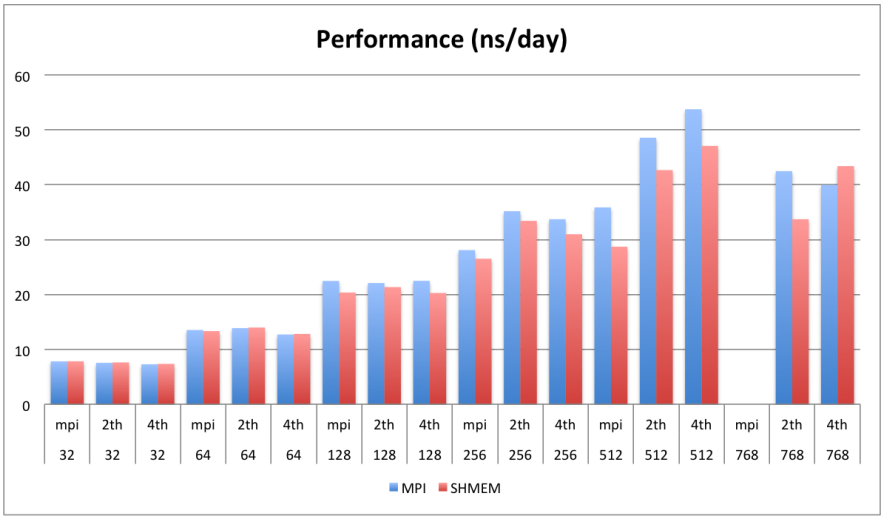


Figure 3: Performance of GROMACS on HECToR Phase 3 for ADH Cubic test case.

which also are affected by imbalance, are included on the SHMEM time. It is worth noting also that the maximum collective does not show up at the top of the profiling report. The heuristics added to reduce the number of overall synchronization improve the performance of the SHMEM implementation. Disabling these heuristics increases the walltime of the SHMEM implementation up to a 20%. This experimental data make us believe that the fine-tuning of these heuristics may improve the performance. In particular, tuning these heuristics for the particular conditions of each execution (maybe by the means of a user variable or configuration) could enhance the performance of the different executions.

3.2 Overall performance

We have evaluated two different benchmark problems to measure the performance of the SHMEM implementation using the HECToR, whose results are shown below.

Figures 3 and 4 show the performance of the GROMACs SHMEM and MPI implemnetations

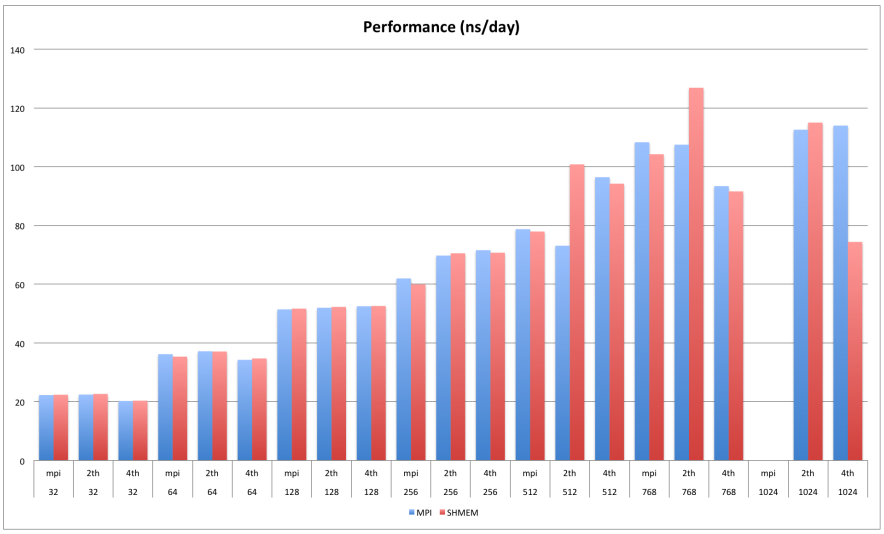


Figure 4: Performance of GROMACS on HECToR Phase 3 for the Grappa 45k test case.

in ns/day (nanoseconds of simulation per running day). This is the real performance of the application, computed as the average performance of three different executions for each number of threads and cores. The ADH testcase shows more imbalance than the Grappa testcase. Its performance is worse using SHMEM as the implementation is more affected by the imbalance due to the synchronization of the size and offset across all ranks during the domain decomposition. This effect hinders the performance compared to the MPI implementation. Performance of the SHMEM implementation in a more-balanced execution, such as the Grappa benchmark in Figure 4, matches the performance of the MPI implementation, and sometimes outperforms it.

4 Conclusions and Future Work

In this work we have presented the experience of porting a real application with a large code base to the SHMEM programming library. We have detailed the different problems encountered and explored the different possible solutions. We believe that this experience will be useful for other programmers interested in porting codes to SHMEM, or to one-side programming models in general.

In general, many decisions in the existing code, aimed at improving the performance of the MPI implementation, limited the performance of SHMEM. For example, some routines compressed data into a buffer in order to produce a single MPI Send Receive call. This is not required in SHMEM, where it would be possible to directly send the data to other ranks in separated put calls with lower overhead.

The critical next step on this work would be to lift the current limitation of using the same number of PP and PME nodes. There are different implementation possibilities, but require a notable development and refactoring effort in the GROMACs communication code.

If the SPMD restrictions were lifted, at least to the point where an application may have separate SHMEM communication sets, the performance would improve notably.

An alternative implementation using MPI one-side communications [4], recently adopted in

the MPI 3.0 standard, may be seen as better suited given the circumstances. The development effort and analysis performed during this project will leverage the implementation cost a future MPI one-side approach, as the major areas of improvement and the refactoring has been already performed.

5 Acknowledgements

This work was funded under the HECToR Distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR A Research Councils UK High End Computing Service - is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE Support Service is provided by NAG Ltd.

References

[1] Carlo Cavazzoni. Eurora: a european architecture toward exascale. In *Proceedings of the Future HPC Systems: the Challenges of Power-Constrained Performance*, FutureHPC '12, pages 1:1–1:4, New York, NY, USA, 2012. ACM.

[2] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.

[3] HECToR. Hector home page, June 2012.

[4] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, D.K. Panda, W. Gropp, and R. Thakur. High performance mpi-2 one-sided communication over infiniband. In *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 531–538, 2004.

[5] Sander Pronk, Szilrd Pll, Roland Schulz, Per Larsson, Pr Bjelkmar, Rossen Apostolov, Michael R. Shirts, Jeremy C. Smith, Peter M. Kasson, David van der Spoel, Berk Hess, and Erik Lindahl. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29(7):845–854, 2013.

[6] Hongzhang Shan, Nicholas J. Wright, John Shalf, Katherine Yelick, Marcus Wagner, and Nathan Wichmann. A preliminary evaluation of the hardware acceleration of the cray gemini interconnect for pgas languages and comparison with mpi. In *Proceedings of the second international workshop on Performance modeling, benchmarking and simulation of high performance computing systems*, PMBS '11, pages 13–14, New York, NY, USA, 2011. ACM.

Flat Combining Synchronized Global Data Structures

Brandon Holt[1], Jacob Nelson[1], Brandon Myers[1], Preston Briggs[1], Luis Ceze[1], Simon Kahan[1,2] and Mark Oskin[1]

[1] University of Washington
[2] Pacific Northwest National Laboratory
{bholt,nelson,bdmyers,preston,luisceze,skahan,oskin}@cs.washington.edu

Abstract

The implementation of scalable synchronized data structures is notoriously difficult. Recent work in shared-memory multicores introduced a new synchronization paradigm called *flat combining* that allows many concurrent accessors to cooperate efficiently to reduce contention on shared locks. In this work we introduce this paradigm to a domain where reducing communication is paramount: distributed memory systems. We implement a flat combining framework for Grappa, a latency-tolerant PGAS runtime, and show how it can be used to implement synchronized global data structures. Even using simple locking schemes, we find that these flat-combining data structures scale out to 64 nodes with 2x-100x improvement in throughput. We also demonstrate that this translates to application performance via two simple graph analysis kernels. The higher communication cost and structured concurrency of Grappa lead to a new form of distributed flat combining that drastically reduces the amount of communication necessary for maintaining global sequential consistency.

1 Introduction

The goal of partitioned global address space (PGAS) [10] languages and runtimes is to provide the illusion of a single shared memory to a program actually executing on a distributed memory machine such as a cluster. This allows programmers to write their algorithms without needing to explicitly manage communication. However, it does not alleviate the need for reasoning about consistency among concurrent threads. Luckily, the PGAS community can leverage a large body of work solving these issues in shared memory and explore how differing costs lead to different design choices.

It is commonly accepted that the easiest consistency model to reason about is *sequential consistency* (SC), which enforces that all accesses are committed in program order and appear to happen in some global serializable order. To preserve SC, operations on shared data structures should be *linearizable* [14]; that is, appear to happen atomically in some global total order. In both physically shared memory and PGAS, maintaining linearizability presents performance challenges. The simplest way is to have a single global lock to enforce atomicity and linearizability through simple mutual exclusion. Literally serializing accesses in this way is typically considered prohibitively expensive, even in physically shared memory. However, even in fine-grained lock-free synchronization schemes, as the number of concurrent accessors increases, there is more contention, resulting in more failed synchronization operations. With the massive amount of parallelism in a cluster of multiprocessors and with the increased cost of remote synchronization, the problem is magnified.

A new synchronization technique called *flat combining* [12] coerces threads to *cooperate* rather than *contend*. Threads delegate their work to a single thread, giving it the opportunity to combine multiple requests in data-structure specific ways and perform them free from

contention. This allows even a data structure with a single global lock to scale better than complicated concurrent data structures using fine-grained locking or lock-free mechanisms.

The goal of this work is to apply the flat-combining paradigm to a PGAS runtime to reduce the cost of maintaining sequentially consistent data structures. We leverage Grappa, a PGAS runtime library optimized for fine-grained random access, which provides the ability to tolerate long latencies by efficiently switching between many lightweight threads as sketched out in prior work [19]. We leverage Grappa's latency tolerance mechanisms to allow many fine-grained synchronized operations to be combined to achieve higher, scalable throughput while maintaining sequential consistency. In addition, we show how a generic flat-combining framework can be used to implement multiple global data structures.

The next section will describe in more detail the Grappa runtime system that is used to implement flat combining for distributed memory machines. We then explain the flat-combining paradigm in more depth and describe how it maps to a PGAS model. Next, we explain how several data structures are implemented in our framework and show how they perform on simple throughput workloads as well as in two graph analysis kernels.

2 Grappa

Irregular applications are characterized by having unpredictable data-dependent access patterns and poor spatial and temporal locality. Applications in this class, including data mining, graph analytics, and various learning algorithms, are becoming increasingly prevalent in high-performance computing. These programs typically perform many fine-grained accesses to disparate sources of data, which is a problem even at multicore scales, but is further exacerbated on distributed memory machines. It is often the case that naively porting an application to a PGAS system results in excessive communication and poor access patterns [8], but this class of applications defies typical optimization techniques such as data partitioning, shadow objects, and bulk-synchronous communication transformations. Luckily, applications in this class have another thing in common: abundant amounts of data access parallelism. This parallelism can be exploited in a number of different ways to improve overall throughput.

Grappa is a global-view PGAS runtime for commodity clusters which has been designed from the ground up to achieve high performance on irregular applications. The key is latency tolerance—long-latency operations such as reads of remote memory can be tolerated by switching to another concurrent thread of execution. Given abundant concurrency, there are opportunities to increase throughput by sacrificing latency. In particular, throughput of random accesses to remote memory can be improved by delaying communication requests and aggregating them into larger packets.

Highly tuned implementations of irregular applications in shared-memory, PGAS, and message-passing paradigms, typically end up implementing similar constructs. Grappa includes these as part of its core infrastructure and simply asks the programmer to express concurrency which it can leverage to provide performance.

Grappa's programming interface, implemented as a C++11 library, provides high-level operations to access and synchronize through global shared memory, and task and parallel loop constructs for expressing concurrency. In addition, the Grappa "standard library" includes functions to manipulate a global heap, stock remote synchronization operations such as *compare-and-swap*, and several synchronized global data structures. These features make it suitable for implementing some next-generation PGAS languages like Chapel [6] and X10 [7]. The following sections will explain the execution model of Grappa and the current C++ programming interface.

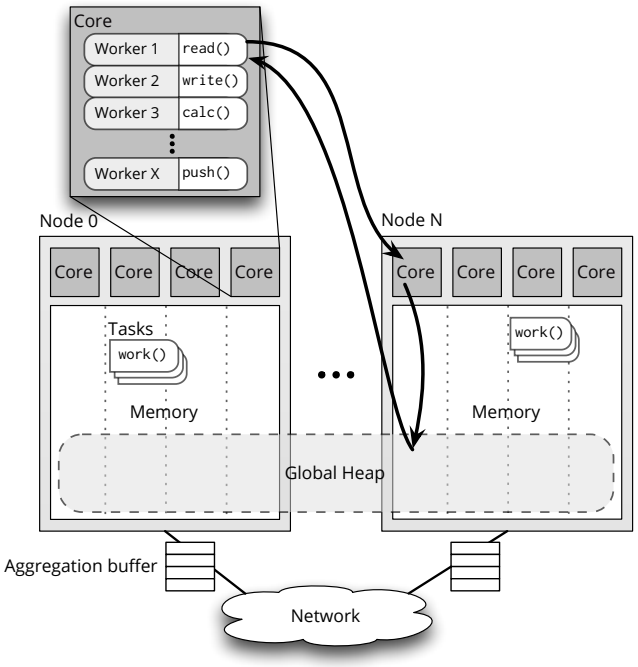


Figure 1: *Grappa System Overview:* Thousands of workers are multiplexed onto each core, context switching to tolerate the additional latency of aggregating remote operations. Each core has exclusive access to a slice of the global heap which is partitioned across all nodes, as well as core-private data and worker stacks. Tasks can be spawned and executed anywhere, but are bound to a worker in order to be executed.

2.1 Tasks and Workers

Grappa uses a task-parallel programming model to make it easy to express concurrency. A *task* is simply a function object with some state and a function to execute. Tasks may block on remote accesses or synchronization operations. The Grappa runtime has a lightweight threading system that uses prefetching to scale up to thousands of threads on a single core with minimal increase in context-switch time. In the runtime, *worker* threads pull these programmer-specified tasks from a queue and execute them to completion. When a task blocks, the worker thread executing it is suspended and consumes no computational resources until woken again by some event.

2.2 Aggregated Communication

The most basic unit of communication in Grappa is an *active message* [24]. To make efficient use of the networks in high-performance systems, which typically achieve maximum bandwidth only for messages on the order of 64 KB, all communication in Grappa is sent via an aggregation layer that automatically buffers messages to the same destination.

2.3 Global Memory

In the PGAS style, Grappa provides a global address space partitioned across the physical memories of the nodes in a cluster. Each core owns a portion of memory, which is divided among execution stacks for the core's workers, a core-local heap, and a slice of the global heap.

All of these can be addressed by any core in the system using a `GlobalAddress`, which encodes both the owning core and the address on that core. Additionally, addresses into the global heap are partitioned in a block-cyclic fashion, so that a large allocation is automatically distributed among many nodes. For irregular applications, this helps avoid hot spots and is typically sufficient for random access.

Grappa enforces strict isolation—all accesses must be done by the core that owns it via a message, even between processes on the same physical memory. At the programming level, however, this is hidden behind higher-level remote operations, which in Grappa are called *delegates*. Figure 1 shows an example of a delegate `read` which blocks the calling task and sends a message to the owner, who sends a reply with the data and wakes the caller.

3 Flat Combining

At the most basic level, the concept of flat combining is about enabling cooperation among threads rather than contention. The benefits can be broken down into three components: improved locality, reduced synchronization, and data-structure-specific optimization. We will explore how this works in a traditional shared-memory system, and then describe how the same concepts can be applied to distributed memory.

3.1 Physically shared memory

Simply by delegating work to another core, locality is improved and synchronization is reduced. Consider the shared synchronous stack shown in Figure 2, with pre-allocated storage and a `top` pointer protected by a lock. Without flat combining, whenever a thread attempts to push something on the stack, it must acquire the lock, put its value into the storage array, bump the top, and then release the lock. When many threads contend for the lock, all but one will fail and have to retry. Each attempt forces an expensive memory fence and consumes bandwidth, and as the number of threads increases, the fraction of successes plummets. Under flat combining, instead, threads add requests to a *publication list*. They each try to acquire the lock, and the one that succeeds becomes the combiner. Instead of retrying, the rest spin on their request waiting for it to be filled. The combiner walks the publication list, performs all of the requests, and when done, releases the lock. This allows the one thread to keep the data structure in cache, reducing thrashing between threads on different cores. It also greatly reduces contention on the lock, but introduces a new point of synchronization—adding to the publication list. However, if a thread performs multiple operations, it can leave its publication record in the list and amortize the synchronization cost. This publication list mechanism can be re-used in other data structures, saving each from needing to implement its own clever synchronization.

The above example of delegation is compelling in itself. However, the crux of prior work is that data structure-specific optimization can be done to perform the combined operations more efficiently. As the combiner walks the publication list, it merges each non-empty publication record into a combined operation. In the case of the stack example shown in Figure 2, as it walks the list, Thread 4 keeps track of the operations on its own temporary stack. When it encounters Thread 2's pop, it recognizes that it can satisfy that pop immediately with the push

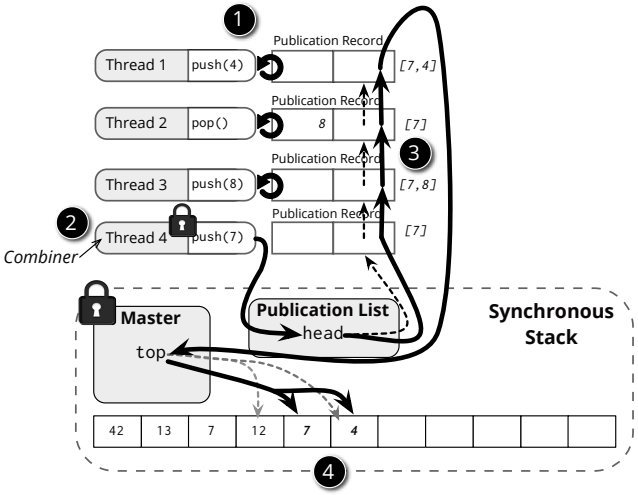


Figure 2: *Flat combining in shared memory.* To access the shared stack, each thread adds its request to the publication list (1). Thread 4 acquires the stack's lock and becomes the combiner (2), while the remaining threads spin waiting for their request to be satisfied. The combiner walks the publication list from the head, matching up Thread 3's push with Thread 2's pop on its own private stack (3). The two remaining pushes are added to the top of the shared stack (4). Finally, the top is updated, and Thread 4 releases the lock and continues its normal execution.

it just processed from Thread 3, so it fills both of their records and allows them to proceed. After traversing the rest of the publication list, the thread applies the combined operation to the actual data structure, in this case, the two unmatched pushes are added to the top of the stack. In the case of the stack, combining came in the form of matched pushes and pops, but many data structures have other ways in which operations can be matched.

3.2 Grappa

In a PGAS setting, and in Grappa in particular, the cost of global synchronization and the amount of concurrency is even greater than in shared memory. With thousands of workers per core, in a reasonably sized cluster there are easily millions of workers. This presents an opportunity for flat combining to pay off greatly, but also poses new challenges.

To illustrate how flat combining can be applied to Grappa, we must first describe what a global data structure looks like. Figure 3 shows a simple PGAS translation of the shared-memory stack from earlier. A storage array is allocated in the global heap, so its elements are striped across all the cores in the system. One core is designated the *master* to enforce global synchronization, and holds the elements of the data structure that all concurrent accessors must agree on, in this case, the `top` pointer.

All tasks accessing the stack hold a `GlobalAddress` to the master object, and invoke custom *delegate methods* that, like the `read` delegate described earlier, block the task until complete. Example code to do a `push` is shown in Figure 5. The task must send a message to the master to acquire the lock. If successful, it follows the `top` pointer, writes its new value to the end of

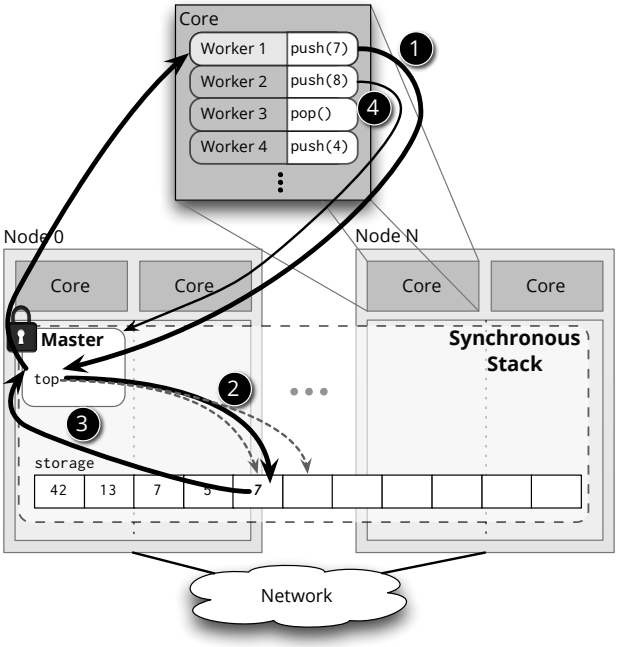


Figure 3: *Stack without combining.* To do its push, Worker 1 sends a message to synchronize with the master on Core 0 (1), which sends another message to write the value to the top of the stack (2), bumps the synchronized `top` pointer (3), and finally continues. Worker 2, and workers on other cores, must block at the master and wait for Worker 1 to complete its push before doing their operations (4).

the stack, returns to bump the `top` pointer and release the lock, and finally sends a message back to wake the calling worker. All others block at the first message until the lock is released. Grappa's user-level threading allows requests to block without consuming compute resources. However, all workers on each core perform this synchronization and serialize on a single core, causing that core to become the bottleneck.

Centralized Combining. A first thought might be to directly apply the idea of flat combining to the serialization at the *master*. The worker that acquires the lock can walk through the requests of all the other workers waiting to acquire the lock and combine them. In the case of the Stack, this would mean matching pushes and pops, applying the remainder, and sending messages back to all remote workers with results, before starting another round of combining. This approach reduces traffic to the data structure storage, but a single core must still process every request, so it cannot scale if every other core is generating requests at the same rate.

Distributed Combining. Instead of all workers sending independent synchronization messages and putting the burden all on one core, each core can instead do its own combining first then synchronize with the master in bulk. Distributing the combining to each core allows the majority of the work to be performed in parallel and without communication.

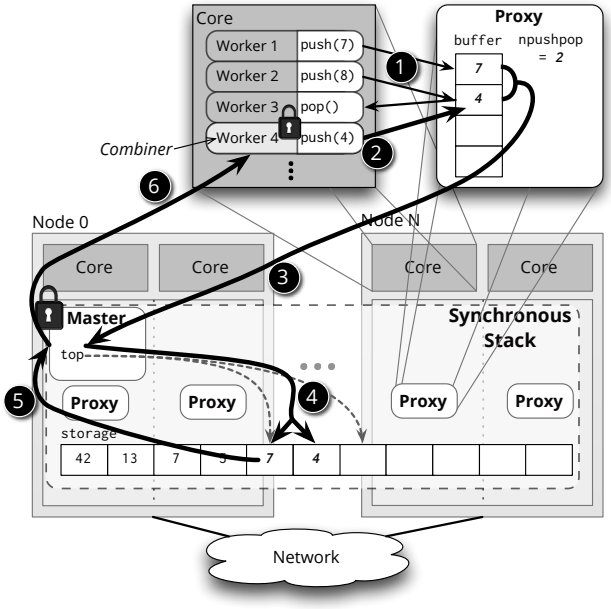


Figure 4: *Stack with distributed combining.* Worker 3's pop matches with Worker 2's push, requiring no global communication (1). After combining locally, Worker 1 and 4's pushes remain, so Worker 4 becomes the core's combiner (2), sends a message to synchronize with the master (3), adds both new values to global stack (4), bumps the top pointer and releases the lock on master (5), and finally wakes Worker 1 (6).

In distributed flat-combining, each core builds its own publication list to track all the operations waiting to be committed to the global data structure. In Grappa, for each global data structure, a local *proxy* object is allocated from the core-local heap to intercept requests. Conceptually, workers add their requests to a local publication list in the proxy, one is chosen to do the combining, and the rest block until their request is satisfied. However, because Grappa's scheduler is non-preemptive, each worker has atomicity "for free" until it performs a blocking operation (such as communication). This means that an explicit publication list with clever synchronization is unnecessary. Instead, workers merge their operations directly into the local proxy, and block, except in restricted cases where they are able to satisfy their requirements immediately without violating ordering rules for consistency (discussed in the next section). The proxy structure is chosen to be able to compactly aggregate operations and efficiently perform matching in cases where it is allowed. Figure 4 shows how pushes and pops are matched on the proxy's local stack.

After all local combining has been done, one requesting worker on the core is chosen to commit the combined operation globally. In Figure 4, Worker 4 becomes the combiner and performs much the same synchronization as in the uncombined case, but is able to push multiple elements with one synchronization. The global lock must still be acquired, so concurrent combined requests (from different cores) must block and serialize on the master, but the granularity of global synchronization is coarser, reducing actual serialization.

Centralized combining could be applied on top of distributed combining, combining the

combined operations at the master node. However, in our experiments, performing the additional combining on the master did not significantly affect performance, so it is left out of our evaluation for space.

Memory Consistency Model. In the context of Grappa, sequential consistency guarantees that within a task, operations will be in task order and that all tasks will observe the same global order. The Grappa memory model is essentially the same as the C++ memory model [5, 4, 15], guaranteeing sequential consistency for data-race free programs. To be conservative, delegate operations block the calling worker until they have become globally visible, ensuring they can be used for synchronization. As such, delegate operations within a task are guaranteed to be globally visible in program order and all tasks observe the same global order. Operations on synchronized data structures must provide the same guarantees. Because it is not immediately obvious that this distributed version of flat combining preserves sequential consistency, we now argue why it does.

To behave in accordance with sequential consistency, operations on a particular data structure must obey a consistent global order. One way to provide this is to guarantee *linearizability* [14] of operations, which requires that the operation appear to take effect globally at some instantaneous point during invocation of the API call. This ensures that a consistent total order can be imposed on operations on a single data structure. For operations to be globally linearizable, first of all the execution of local combined operations must be serializable; this is unchanged from shared-memory flat-combining, and is trivially true due to the atomicity enabled by cooperative multithreading. Second, combined operations must be committed atomically in some globally serializable order. When committing, combined operations are serialized at the particular core that owns the synchronization (the *master* core for the Stack). Whenever a global commit starts, a fresh combiner must be created for subsequent operations to use. If they were to use the old combiner's state to satisfy requests locally, it would violate global ordering because the local object would not reflect other cores' updates. For this serial "concatenation" of serialized batches to be valid, the order observed by workers during local combining must be the same as what can be observed globally as operations are committed.

In the case of a stack or queue, this guarantee comes from applying a batch of push or pop operations atomically in a contiguous block on the global data structure. Matching pushes and pops locally for the Stack is one exception to the rule that operations must block until globally committed. Because a pop "destroys" the corresponding push, these operations together are independent of all others and can conceptually be placed anywhere in a global order. It is trivial to respect program order with this placement, so they can be safely matched locally.

Combining set and map operations exposes more nuances in the requirements for consistency. Insert and lookup operations performed by different tasks are inherently unordered unless synchronized externally. Therefore, a batch of these operations can be committed globally in parallel, since they are guaranteed not to conflict with each other. Note that if an insert finds that its key is already in the combiner object, it does not need to send its own message. However, it must still block until that insert is done to ensure that a subsequent operation it performs cannot be reordered with it, preserving program order. Similarly, lookups can piggy-back on other lookups of the same key.

Using intuition from store/write buffers in modern processors, it is tempting to allow a lookup to be satisfied locally by inspecting outstanding inserts. However, this would allow the local order in which keys were inserted to be observed. Then to preserve SC, that same order would need to be respected globally, forcing each batch to commit atomically with respect to other cores' batches. Enforcing this would be prohibitively expensive, so a cheaper solution is

chosen: lookups only get their values from the global data structure, therefore batches can be performed in parallel.

These requirements only guarantee linearizability of operations on a single data structure. To guarantee sequential consistency with respect to all accesses, data-race freedom must be guaranteed by the program, as in the C++ memory model for shared memory.

4 Grappa FC Framework

To leverage the flat-combining paradigm in Grappa, we implemented a generic framework to improve performance for a number of global data structures. The FC framework handles the common problems of managing the combiners, handling worker wake-ups, and maintaining progress. When hooking into the framework, each data structure need only define how to combine operations and globally commit them.

As mentioned before, the Grappa FC framework takes a different approach than the original flat-combining work for expressing how operations combine. For each global structure instance, a *proxy* object is created on each core and each worker merges its request into that structure before blocking or becoming the combiner. Each global data structure must define a *proxy* that has *state* to track combined requests, *methods* to add new requests, and a way to *sync* the state globally. An example proxy declaration for the GlobalStack is shown in Figure 5.

The FC framework is responsible for ensuring that all of the combined operations eventually get committed. There are a number of ways progress could be guaranteed, but one of the simplest is to ensure that as long as there are any outstanding requests, at least one worker is committing a combined operation. When that combined operation finishes, if there are still outstanding requests that have built up in the meantime, another blocked worker is chosen to do another combined synchronization.

While a combining operation is in flight, new requests continue to accumulate. The framework transparently wraps the proxy object so when one `sync` starts, it can direct requests to a fresh combiner. This is done by hiding instances of proxy objects behind a C++ smart-pointer-like object which provides the pointer to the current combiner, and whenever it detects that it should send, allocates a new combiner and points subsequent references to that.

4.1 Global Stack and Queue

Figure 5 shows an excerpt from the definition of the proxy object for the Grappa GlobalStack. The proxy tracks pushes in the `pushed_values` array. When `pop` is called, if there are pushes, it immediately takes the top value and wakes the last pusher. Otherwise, it adds a pointer to a location on its stack where the `sync` operation will write the result. Because of local matching, when a Stack proxy is synchronized, it will have either all pops or all pushes, which makes the implementation of sync straightforward. Note that the operation to synchronize a batch of pushes looks almost identical to the code to do a single push from Figure 5.

The GlobalQueue has nearly the same implementation as the stack, but is unable to match locally.

4.2 Global Set and Map

The Grappa GlobalSet uses a simple chaining design, implemented with a global array of cells (allocated from the global heap), which are partitioned evenly among all the cores, and indexed with a hash function. Our flat-combining version supports both `insert` and `lookup`. To track

```
// Global master state
class GlobalStack {
  GlobalAddress<T> top;
  Grappa::Mutex lock;
};
// Push *without* combining
void push(GlobalAddress<GlobalStack> master, T e){
  // from stack's master core, perform write to top and increment atomically
  delegate::call(master.core(), [master,e]{
    master->lock.acquire();
    delegate::write(master->top, e);
    master->top++;
    master->lock.release();
  }); // blocks until response arrives
}
// Definition of proxy
class GlobalStackProxy : Grappa::FCProxy {
  GlobalAddress<GlobalStack> master;
  // Local state for tracking requests
  T  pushed_values[1024];
  T* popped_results[1024];
  int npush, npop;

  // Combining Methods
  void push(T val);
  T pop();

  // Global sync (called by FC framework)
  void sync() override {
    if (npush > 0) {
      // on master: acquire lock, return top ptr
      auto top = delegate::call(master.core(),[=]{
        master->lock.acquire();
        return master->top;
      });
      // copy values to top of stack
      Grappa::memcpy(top, pushed_values, npush);
      // bump top and release lock
      delegate::call(master.core(),[master,npush]{
        master->top += npush;
        master->lock.release();
      });
    } else if (npop > 0) {} // elided for space...
  }
};
```

Figure 5: Snippet of code from GlobalStack.

all of the keys waiting to be inserted, we use the hash set implementation from the C++11 standard library (`std::unordered_set`). As with pops, lookups must provide pointers to stack space where results should be put, which is done with a hash map (again from the C++ standard library) from keys to lists of pointers. As discussed in Section 3.2, matching lookups with inserts locally would force a particular sequential order. Instead, we only allow matching inserts with inserts and lookups with lookups, allowing `sync` to simply issue all inserts and lookups in parallel and block until all have completed.

Our implementation of the GlobalMap matches that of the Set but of course stores values.

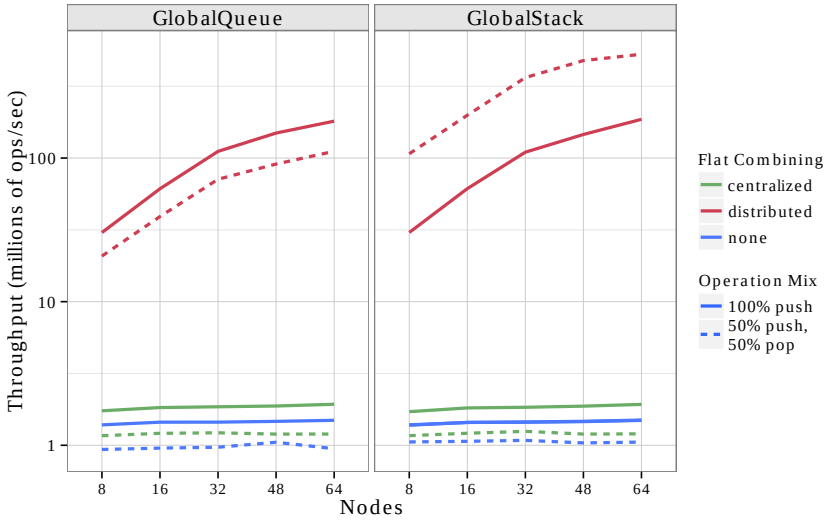


Figure 6: *Queue and Stack performance.* Results are shown on a log scale for a throughput workload performing 256 million operations with 2048 workers per core and 16 cores per node. Local flat combining improves throughput by at least an order of magnitude and allows performance to scale. Matching pushes and pops enables the stack to perform even better on a mixed workload.

5 Evaluation

To evaluate the impact flat combining has, we ran a series of experiments to test the raw performance of the data structures themselves under different workloads, and then measure the impact on performance of two graph benchmarks. Experiments were run on a cluster of AMD Interlagos processors. Nodes have 32 2.1-GHz cores in two sockets, 64GB of memory, and 40Gb Mellanox ConnectX-2 InfiniBand network cards connected via a QLogic switch.

5.1 Data Structure Throughput

First we measured the raw performance of the global data structures on synthetic throughput workloads. In each experiment, a Grappa parallel loop spawns an equal number of tasks on all cores. Each task randomly chooses an operation based on the predetermined "operation mix," selecting either a push or pop for the Stack and Queue, or an insert or lookup for the Set and Map.

Queue and Stack. Figure 6 shows the results of the throughput experiments for the global Stack and Queue. Results are shown with flat combining completely disabled, only combining at the master core ("centralized"), and combining locally ("distributed").

Despite Grappa's automatic aggregation, without combining, both the stack and queue completely fail to scale because all workers' updates must serialize. Though centralized combining alleviates some of the serialization, its benefit is limited because all operations involve synchronization through a single core. However, with local flat combining, synchronization is done mostly in parallel, with less-frequent bulk synchronization at the master.

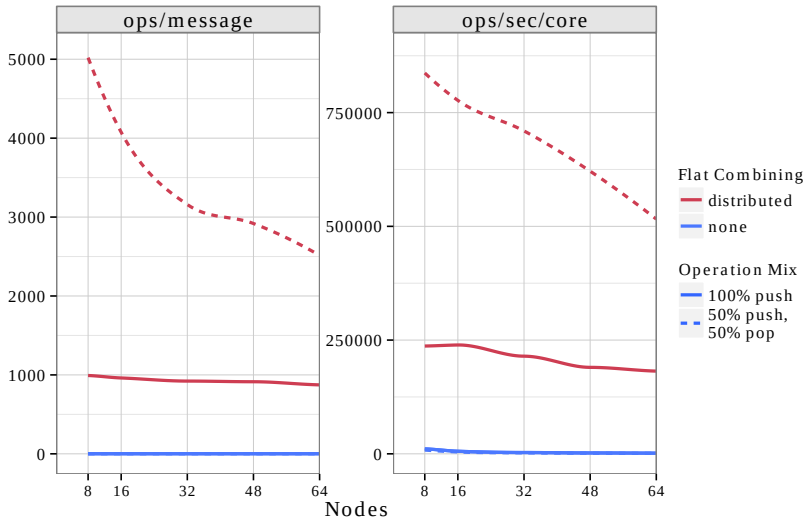


Figure 7: *GlobalStack Statistics.* Measured number of combined messages sent by the Stack with a fixed combining buffer of 1024 elements. Matched pushes and pops result in ops/message being greater than the buffer size.

On the mixed workload, the stack is able to do matching locally, allowing it to reduce the amount of communication drastically, greatly improving its performance. Figure 7 corroborates this, showing that the amount of combining that occurs directly correlates with the observed throughput.

The queue benefits in the same way from reducing synchronization and batching, and its all-push workload performs identically to the stack's. However, the queue is unable to do matching locally, and in fact, the mixed workload performs worse because the current implementation serializes combined pushes and combined pops. This restriction could be lifted with more careful synchronization at the master core.

HashSet and HashMap. Figure 8 shows the throughput results for the Set and Map. Both data structures synchronize at each hash cell, which allows them to scale fairly well even without combining. However, after 32 nodes, scaling drops off significantly due to the increased number of destinations. Combining allows duplicate inserts and lookups to be eliminated, so performs better the smaller the key range. This reduction in message traffic allows scaling out to 64 nodes.

5.2 Application Kernel Performance

The Grappa data structures are synchronized to provide the most general use and match the expectations of programmers and algorithms. In these evaluations, we compare the flat-combining structures against custom, tuned versions that leverage relaxed consistency needs of the applications.

Breadth-First Search. The first application kernel is the Graph500 Breadth-First-Search (BFS) benchmark [11]. This benchmark does a search starting from a random vertex in a

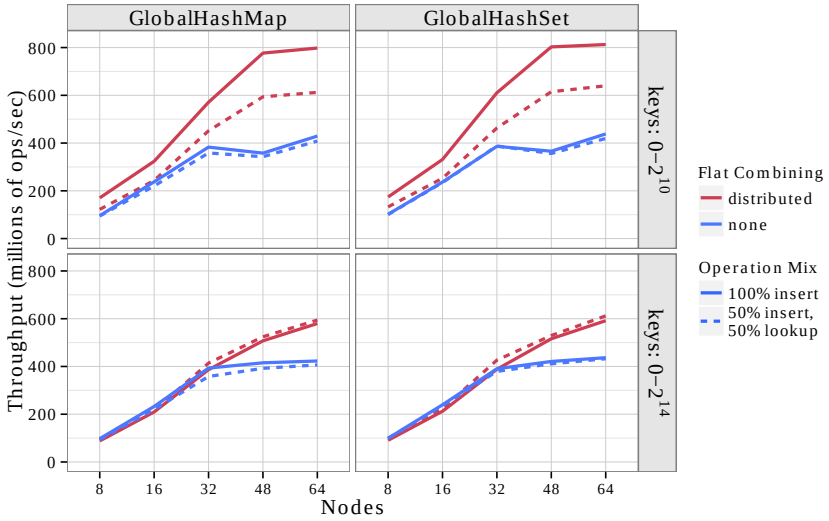
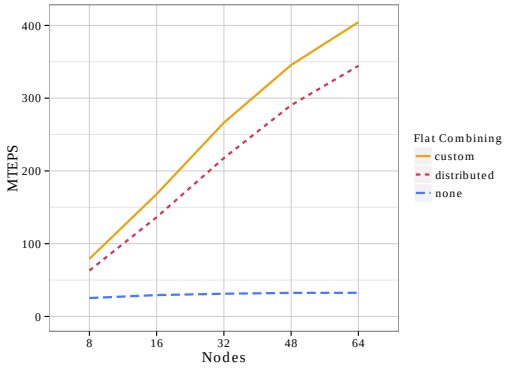


Figure 8: *GlobalHashSet and GlobalHashMap.* Results are shown for a throughput workload inserting and looking up 256 million random keys in a particular range into a global hash with the same number of cells, with 2048 workers per core and 16 cores per node.

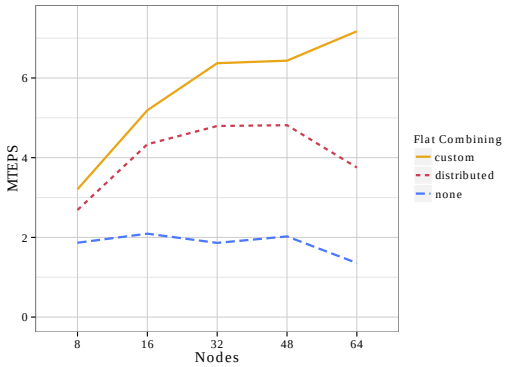
synthetic graph and builds a search tree of parent vertices for each vertex traversed during the search. The BFS algorithm contains a global queue which represents the frontier of vertices to be visited in each level. Our implementation employs the direction-optimizing algorithm by Beamer et al. [2]. The frontier queue is write-only in one phase and read-only in the next, making it amenable to relaxed consistency. We compare performance of BFS using the Grappa FC queue described above with a highly tuned Grappa implementation that uses a custom asynchronous queue.

Figure 9a shows the scaling results. Using the simple queue without flat combining is completely unscalable, but with FC, it tracks the tuned version. This illustrates that providing a safe, synchronized data structure for initially developing algorithms for PGAS is possible without sacrificing scaling.

Connected Components. Connected Components (CC) is another core graph analysis kernel that illustrates a different use of global data structures in irregular applications. We implement the three-phase CC algorithm [3] which was designed for the massively parallel MTA-2 machine. In the first phase, parallel traversals attempt to claim vertices and label them. Whenever two traversals encounter each other, an edge between the two roots is inserted in a set. The second phase performs the classical Shiloach-Vishkin parallel algorithm [22] on the reduced graph formed by the edge set from the first phase, and the final phase propagates the component labels out to the full graph. Creation of the reduced edge set dominates the runtime of this algorithm, but includes many repeated inserts at the boundary between traversals, so is a prime target for the flat-combining Set. Similar to BFS, the Set is write-only first, then read-only later, so further optimizations involving relaxation of consistency can be applied. Therefore, we compare our straightforward implementation using the generic Set with and without flat combining against a tuned asynchronous implementation.



(a) *BFS* with the direction-optimizing algorithm.



(b) *Connected Components* using Kahan's algorithm.

Figure 9: Performance of graph kernels on a Graph500-spec graph of scale 26 (64 million vertices, 1 billion edges). Performance for both is measured in Millions of Traversed Edges Per Second (MTEPS).

The results in Figure 9b show that none of these three scale well out to 64 nodes. However, performing combining does improve the performance over the uncombined case. The tuned version outperforms the synchronous version because it is able to build up most of the set locally on each core before merging them at the end of the first phase. An implementation that did not provide synchronized semantics could potentially relax consistency in a more general way.

6 Related Work

Though much attention has gone to lock-free data structures such as the Treiber Stack [23] and Michael-Scott Queue [18], a significant body of work has explored ways of scaling globally locked data structures using combining to reduce synchronization and hot-spotting. Techniques differ mainly in the structures used to do combining: fixed trees [25], dynamic trees (or "funnels") [20, 17], and randomized trees [1]. In particular, MAMA [17], built for the MTA-2,

leveraged funnels to better utilize hardware optimized for massive concurrency. On the other hand, *flat* combining [12] observed that in multicore systems, hierarchical schemes have too much overhead, and a cleverly implemented *flat* publication list can perform better. Follow-on work introduced parallel flat-combining [13], in which multiple publication lists are combined in parallel and their remainders serialized. Flat combining was also applied in the NUMA (non-uniform memory access) domain with scalable hierarchical locks [9], which improves on prior work on hierarchical locking by leveraging flat-combining's publication mechanism which amortizes the cost of synchronization.

Our work extends the flat-combining paradigm further, to the PGAS domain, where only software provides the illusion of global memory to physically distributed memories. The relatively higher cost of fine-grained data-driven access patterns in PGAS makes flat combining even more compelling. In addition, the physical isolation per node combined with Grappa's engineered per-core isolation guarantees, create a novel situation for combining, which led to our *combining proxy* design.

In the distributed memory and PGAS domains, implementing scalable and distributed synchronization is the name of the game. Besides Grappa, other work has proposed leveraging latency tolerance in PGAS runtimes, including MADNESS [21] which proposes ways of using asynchronous operations in UPC, and Zhang et al. [26] who use asynchronous messages in a UPC Barnes-Hut implementation to overlap computation with communication and allow for buffering. Jose et al. introduced UPC Queues [16], a library which provides explicit queues as a replacement for locks to synchronize more efficiently and allow for buffering communication. This work demonstrates a similar use case for synchronized queues as in our work, and uses them with the Graph500 benchmark as well. In contrast, Grappa performs much of the same message aggregation invisibly to the programmer, and the FC framework can be used to implement many data structures.

7 Conclusion

Coming from the multi-core domain, the flat-combining paradigm provides a new perspective to global synchronization, bringing with it both challenges and new opportunities. We have shown that the additional concurrency that comes with a latency-tolerant runtime, rather than compounding the problem, provides a new opportunity for reducing communication by combining locally. In PGAS implementations there is typically a large discrepancy between the first simple description of an algorithm and the final optimized one. Our distributed flat-combining framework allows easy implementation of a library of flat-combined linearizable global data structures, allowing even simple applications that use them to scale out to a thousand cores and millions of concurrent threads.

References

[1] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Euro-Par 2010-Parallel Processing*, pages 151–162. Springer, 2010.

[2] S. Beamer, K. Asanovi, and D. Patterson. Direction-optimizing breadth-first search. In *Conference on Supercomputing (SC-2012)*, November 2012.

[3] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–14. IEEE, 2007.

[4] H.-J. Boehm. A Less Formal Explanation of the Proposed C++ Concurrency Memory Model. C++ standards committee paper WG21/N2480 = J16/07-350, http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2480.html, December 2007.

[5] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.

[6] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel Language. *International Journal of High Performance Computing Application*, 21(3):291–312, Aug. 2007.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[8] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 36–47. ACM, 2005.

[9] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining NUMA locks. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 65–74, New York, NY, USA, 2011. ACM.

[10] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, Inc., Hoboken, NJ, USA, 2005.

[11] Graph 500. http://www.graph500.org/, July 2012.

[12] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 355–364. ACM, 2010.

[13] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Scalable flat-combining based synchronous queues. In *Distributed Computing*, pages 79–93. Springer, 2010.

[14] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[15] ISO/IEC JTC1/SC22/WG21. ISO/IEC 14882, Programming Language - C++ (Committee Draft). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2800.pdf, 2008.

[16] J. Jose, S. Potluri, M. Luo, S. Sur, and D. Panda. UPC Queues for scalable graph traversals: Design and evaluation on Infiniband clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, 2011.

[17] S. Kahan and P. Konecny. MAMA!: A memory allocator for multithreaded architectures. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 178–186, New York, NY, USA, 2006. ACM.

[18] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275. ACM, 1996.

[19] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching large graphs with commodity processors. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism*, HotPar'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[20] N. Shavit and A. Zemach. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

[21] A. Shet, V. Tipparaju, and R. Harrison. Asynchronous programming in UPC: A case study and potential for improvement. In *Workshop on Asynchrony in the PGAS Model*. Citeseer, 2009.

[22] Y. Shiloach and U. Vishkin. An O(N log(N)) parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.

[23] R. K. Treiber. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[24] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM.

[25] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *Computers, IEEE Transactions on*, 100(4):388–395, 1987.

[26] J. Zhang, B. Behzad, and M. Snir. Optimizing the Barnes-Hut algorithm in UPC. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 75. ACM, 2011.

Applying Type Oriented Programming to the PGAS Memory Model

Nick Brown

Abstract

The Partitioned Global Address Space memory model has been popularised by a number of languages and applications. However this abstraction can often result in the programmer having to rely on some in built choices and with this implicit parallelism, with little assistance by the programmer, the scalability and performance of the code heavily depends on the compiler and choice of application.

We propose an approach, type oriented programming, where all aspects of parallelism are encoded via types and the type system. The type information associated by the programmer will determine, for instance, how an array is allocated, partitioned and distributed. With this rich, high level of information the compiler can generate an efficient target executable. If the programmer wishes to omit detailed type information then the compiler will rely on well documented and safe default behaviour which can be tuned at a later date with the addition of types.

The type oriented parallel programming language Mesham, which follows the PGAS memory model, is presented. We illustrate how, if so wished, with the use of types one can tune all parameters and options associated with this PGAS model in a clean and consistent manner without rewriting large portions of code. An FFT case study is presented and considered both in terms of programmability and performance - the latter we demonstrate by a comparison with an existing FFT solver.

1 Introduction

As the problems that the HPC community looks to solve become more ambitious then the challenge will be to provide programmers, who might be non HPC experts, with usable and consistent abstractions which still allow for scalability and performance. Partitioned Global Address Space is a memory model providing one such abstraction and allows for the programmer to consider the entire system as one entire global memory space which is partitioned and each block local to some process. Numerous languages and frameworks exist to support this model but all, operating at this higher level, impose some choices and restrictions upon the programmer in the name of abstraction.

This paper proposes a trade-off between explicit parallelism, which can yield good performance and scalability if used correctly, and implicit parallelism which promotes simplicity and maintainability. Type oriented programming addresses the issue by providing the options to the end programmer to choose between explicit and implicit parallelism. The approach is to design new types, which can be combined to form the semantics of data governing parallelism. A programmer may choose to use these types or may choose not to use them and in the absence of type information the compiler will use a well-documented set of default behaviours. Additional type information can be used by the programmer to tune or specialise many aspects of their code which guides the compiler to optimise and generate the required parallelism code. In short these types for parallelisation are issued by the programmer to instruct the compiler to perform the expected actions during compilation and in code generation. They are predefined by expert HPC programmers in a type library and used by the application programmer who many not have specialist HPC knowledge.

Programmer imposed information about parallelism only appears in types at variable declaration and type coercions in expressions and assignments. A change of data partition or communication pattern only require a change of data types, while the traditional approaches may require rewriting the entire structure of the code. A parallel programming language, Mesham which follows the PGAS memory model, has been developed which follows this paradigm and we study a Fast Fourier Transformation (FFT) case study written in Mesham to evaluate the proposed approach. The pursuit for performance and scalability is a major objective of HPC and we compare the FFT Mesham version with that of an existing, mature solving framework and also consider issues of programmability.

2 The Challenge

The difficulty of programming has been a challenge to parallel computing over the past several decades[8]. Whilst numerous languages and models have been proposed, they mostly suffer from the same fundamental trade-off between simplicity and expressivity. Those languages which abstract the programmer sufficiently to allow for conceptual simplicity often far remove the programmer from the real world execution and impose upon them predefined choices such as the method of communication. The parallel programming solutions which provide the programmer with full control over their code often result in great amounts of complexity which can be difficult for even expert HPC programmers to master for non-trivial problems, let alone the non-expert scientific programmers which often require HPC.

PGAS languages, which provide for the programing memory model abstraction of a global address space which is partitioned and each portion local to a process also suffers from this trade off. For instance, to achieve this memory model the programmer operates at a higher level far removed from the actual hardware and often key aspects, such as the form of data communication, are abstracted away with the programmer having no control upon these key attributes. Operating in a high level environment, without control of lower level decisions, can greatly affect performance and scalability of codes with the programmer reliant on the compiler "making the right choice" when it comes to some critical aspects of parallelism.

Whilst the PGAS memory abstraction is a powerful one, on its own it still leaves complexity to the end programmer in many cases. For example changing the distribution of data amongst the processes can still require the programmer to change numerous aspects of their code.

3 Type oriented programming

The concept of a type will be familiar to many programmers. A large subset of languages follow the syntax *Type Variablename*, such as *int a* or *float b*, which is used to declare a variable. Such statements affect both the static and dynamic semantics - the compiler can perform analysis and optimisation (such as type checking) and at runtime the variable has a specific size and format. It can be thought that the programmer provides information, to the compiler, via the type. However, there is only so much that one single type can reveal, and so languages often include numerous keywords in order to allow for the programmer to specify additional information. Using the C programming language as an example, in order to declare a variable m to be a read only character where memory is allocated externally, the programmer writes *extern const char m*. Where *char* is the type and both *extern* and *const* are inbuilt language keywords. Whilst this approach works well for sequential languages, in the parallel programming domain there are potentially many more attributes which might need to be associated; such as where the data

is located, how it is communicated and any restrictions placed upon this. Representing such a rich amount of information via multiple keywords would not only bloat the language, it might also introduce inconsistencies when keywords were used together with potentially conflicting behaviours.

Instead our approach is to allow for the programmer to encode all variable information via the type system, by combining different types together to form the overall meaning. For instance, *extern const char m* becomes *var m:Char::const::extern*, where *var m* declares the variable, the operator *:* specifies the type and the operator *::* combines two types together. In this case, a **type chain** is formed by combining the types *Char*, *const* and *extern*. Precedence is from right to left where, for example, the read only properties of the *const* type override the default read & write properties of *Char*. It should be noted that some type coercions, such as *Int::Char* are meaningless and so rules exist within each type to govern which combinations are allowed.

Within type oriented programming the majority of the language complexity is removed from the core language and instead resides within the type system. The types themselves contain their specific behaviour for different usages and situations. The programmer, by using and combining types, has a high degree of control which is relatively simple to express and modify. Not only this, the high level of type information provides a rich amount of information upon which the compiler can use and optimise the code. In the absence of detailed type information the compiler can apply sensible, well documented, default behaviour and the programmer can further specialise this using additional types if required at a later date. The result is that programmers can get their code running and then further tune if needed by using additional types.

Benefits of writing type oriented parallel codes are as follows:

1. **Simplicity** - by providing a well documented, clean, type library the programmer can easily control all aspects of parallelism via types or rely on default well-documented behaviour.

2. **Efficiency** - due to the rich amount of high level information provided by the programmer the compiler can perform much optimisation upon the code. The behaviour of types can control the tricky, low level, details which are essential to performance and can be implemented by domain experts which are then used by non-expert parallel programmers.

3. **Flexibility** - often initial choices made, such as the method of data decomposition, can retrospectively turn out to be inappropriate. However, if one is not careful these choices can be difficult to change once the code has matured. By using types the programmer can easily change fundamental aspects by modifying the type with the compiler taking care of the rest. At a language level, containing the majority of the language complexity in a loosely coupled type library means that adding, removing or modifying the behaviour of types has no language wide side effect and the "core" language is kept very simple.

4. **Maintainability** - the maintainability of parallel code is essential. Current production parallel programs are often very complex and difficult to maintain. By providing for simplicity and flexibility it is relatively simple for the code to be modified at a later stage.

4 Mesham

A parallel programming language, Mesham[1], has been created based around an imperative programming language with extensions to support the type oriented concept. By default the

language follows the Partitioned Global Address Space memory model where the entire global memory, which is accessible from every process, is partitioned and each block has an affinity with a distinct process. Reading from and writing to memory (either local or another processes' chunk) is achieved via normal variable access and assignment. By default, in the absence of further types, communication is one sided but this can be overridden using optional additional type information.

The language itself has fifty types in the external type library. Around half of these are similar in scope to the types introduced in the previous section and other types are more complex allowing one to control aspects such as explicit communication, data composition and data partitioning & distribution. In listing 1 the programmer is allocating two integers, a and b on lines one and two respectively. They exist as a single copy in global memory and variable a is held in the memory of process zero, b is in the memory associated with process two. At line three the assignment (using operator $:=$ in Mesham) will copy the value held in b at process two into variable a which resides in the memory of process zero. In the absence of any further type information the communication associated with such an assignment is one-sided, which is guaranteed to be safe and consistent but might not be particularly performant.

```
1 var a : Int :: allocated[single[on[0]]];
2 var b : Int :: allocated[single[on[2]]];
3 a:=b;
```

Listing 1: Default one sided communication

The code in listing 2 looks very similar to that of listing 1 with one important modification, at line one the type *channel* has been added into the type chain of variable a. This type will create an explicit point to point communication link between process two and zero which means that any assignments involving variable a between these processes will use the point to point link rather than one-sided. By default the *channel* type is blocking and control flow will pause until the data has been received by the target process; the programmer could further specialise this to use asynchronous (non-blocking) communication by appending the *async* type into variable a's type chain. In such, asynchronous, cases the semantics of the language is such that the programmer issues explicit synchronisation points, either targeted at a specific variable or all variables, where it is guaranteed that outstanding asynchronous communications will be completed. It can be seen that in the tuning discussed here the programmer, using additional type information, guides the compiler to override the default behaviour. This can be done retrospectively once their parallel code is working and allows one to tune certain aspects which might be crucial to performance or scalability.

```
1 var a : Int :: allocated[single[on[0]] :: channel[2,0];
2 var b : Int :: allocated[single[on[2]]];
3 a:=b;
```

Listing 2: Override communication to blocking point to point

The code examples considered in this section demonstrate that, following the traditional PGAS memory model, using types one can either rely on the simple, safe and well documented default behaviour, or associate additional information and override the defaults as required. Types used to specialise the behaviour are themselves responsible for their specific actions. The benefit of this is that by keeping the majority of the language complexity in the types contained within a loosely coupled type library, it not only results in a much simpler "core" language but also experts can architect types which simply plug into the language.

4.1 Comparison

Unified Parallel C (UPC)[2] is an extension to C designed for parallelism and follows the PGAS memory model. It does this with the addition of language keywords, such as *shared* to marked shared variables, and functions. Due to the limited nature of associating attributes to data using keywords there are still decisions which the UPC programmer is stuck with such as one-sided communication and the programmer is reliant upon the compiler to do the best job it can of optimisation in this regard. Additionally, whilst the memory model is global and communication abstracted, the programmer is still stuck with having to work with low level concepts such as pointers. As discussed, in the type oriented programming model, many additional attributes can be associated with variables by the programmer if the defaults are not suitable. All this type information supports a higher level view of the code because the types controls the behaviour of variables and allows for the elimination of many function calls which are common in more traditional approaches.

High Performance Fortran(HPF)[4] is a parallel extension of Fortran90. The programmer specifies just the data partitioning and allocation, with the compiler responsible for the placement of computation and communication. The type oriented approach differs because programmer can, via types, control far more aspects of parallelism. Alternatively, if not provided, the type system allows for a number of defaults to be used instead. Co-array Fortran (CAF)[6] provides the programmer with a greater degree of control than in HPF, but still the method of communication is implicit and determined by the compiler whilst synchronisations are explicit. CAF uses syntactically shorthanded communication commands like Y[:]=X and synchronisation statements. Having these commands hard wired into the language is popular, not just with CAF but many other parallel languages too, the result is less flexible and more difficult to implement.

Titanium[3] is a PGAS extension to the Java programming language. The PGAS memory model is followed as the implicit model but also allows the programmer to use explicit message passing constructs by using additional language facilities. In this respect, providing for both a higher level implicit memory model and more detailed explicit message passing model, Titanium has some similarities to Mesham. However explicit control in Titanium relies on the programmer issuing in built language keywords such as *broadcast E from p* and/or object methods which results in language bloat. In Titanium moving from the default PGAS memory model to the more explicit message passing requires rewriting portions of the code, whereas with our approach the programmer just needs to modify the type which directs the compiler as to the appropriate way of handing communication. The Mesham type system is designed such that it allows the compiler to generate all possible communication options just by using additional types.

Chapel[7] has been designed, similar to Mesham and Titanium, to allow the programmer to express different abstractions of parallelism. It does this by providing higher and lower levels of abstractions which support automating the common forms of parallel programming via the former and the optimisation and tuning of specific factors using the later. There are some critical differences between Mesham and Chapel. Firstly, many of these higher level constructs in Chapel, such as a reduction is implemented via an inbuilt operator, instead in Mesham these would be types in an independent library. In Chapel, if one declares a single data variable and then writes to it from multiple parallel processes at the same time then this can result in a race condition. The solution is to use a synchronisation variable, via the *sync* keyword in the variables declaration. In the type based approach the Mesham programmer would be using a *sync* type, instead of an inbuilt language keyword, one benefit of this is that if multiple synchronisation constructs were being used (such as Chapel's *sync*, *single* and *atomic* keywords)

then the behaviour in a type chain where precedence is from right to left is well defined. Whilst languages such as Chapel might disallow combinations of these keywords, supporting them in a type chain allows for the programmer to mix the behaviours of different synchronisations in a predicable manner which might be desirable.

5 FFT case study

FFTs are of critical importance to a wide variety of scientific applications ranging from digital signal processing to solving partial differential equations. Parallelised 2D Fast Fourier Transformation (FFT) code is far more complicated than the equivalent sequential code. Direct message passing programming requires the end programmer to handle every detail of parallelisation including writing the appropriate communication commands, synchronizations, and correct index expressions that delimit the range of every partitioned array slice. Whilst using the PGAS memory model can help abstract some of these details the programmer is reliant upon assumptions imposed, in the name of abstraction, which can be costly in terms of scalability and-or performance with other aspects such as the details of data transposition still needing to be considered. A small change of how the data is partitioned or distributed may result in code rewriting. Orienting parallelism around types, however, can relieve the end programmer from writing low level details of parallelisation if these can be derived from the type information in code.

```
1  var n:=8192;
2  var p:=processes() * 2;
3  var i,j;
4
5  var S : array[complex,n,n]::allocated[row[]::single[0]];
6  var A : array[complex,n,n]::allocated[row[]::horizontal[p]::single[
       evendist[]]];
7  var B : array[complex,n,n]::allocated[col[]::horizontal[p]::single[
       evendist[]]];
8  var C : array[complex,n,n]::allocated[row[]::vertical[p]::single[evendist
       []]]::share[B];
9
10 var sins : array[complex,n/2]::allocated[multiple[]];
11 computeSin(sins);
12 proc 0 {readfile(S, "image.dat")};
13
14 A:=S;
15
16 for j from 0 to A.localblocks - 1 {
17         var bid:=A.localblockid[j];
18         for i from A[bid].low to A[bid].high FFT(A[bid][i - A[bid].low],
               sins);
19 };
20
21 B:=A;
22
23 for j from 0 to C.localblocks - 1 {
24         var bid:=C.localblockid[j];
25         for i from C[bid].low to C[bid].high FFT(C[bid][i-C[bid].low],sins
               );
```

```
26 };
27
28 S:=C;
29 proc 0 {writefile(S, "image.dat")};
```

Listing 3: 2D parallel FFT Mesham code

Listing 3 is the parallel aspects of the 2D FFT case study implemented in Mesham. For brevity the actual FFT computation algorithm, a CooleyTukey implementation, and other miscellaneous functions have been omitted. At line 5 the two dimensional array S is declared to comprise of complex numbers be of size n in each dimension, allocated row major fashion and a single copy of it resides upon process zero. This array is used to hold the initial data, an image which is read in at line 12 by process zero and then the results of the transform are placed into it and written back out at line 29. Line 6 declares variable A, again n by n complex numbers, but this time it is partitioned via the *horizontal* type into p distinct partitions which are evenly distributed amongst the processes using the *evendist* type. This even distribution follows a cyclical approach where partitioned blocks will be allocated to process after process and can cycle around if there are more blocks than processes. Line 7 declares the 2D array B to be sized, partitioned and distributed in a similar manner to that of A but this array is indexed column major. The last partitioned array to be declared, C which uses vertical partitioning rather than horizontal, shares the underlying memory with B; in effect this is a different view or abstraction of some existing memory.

Line 10 declares the sinusoid array. Using the *multiple* type without further information results in allocation to the memory of all processes and this is used to compute the pre-calculated constant sinusoid parameters needed by the FFT kernel. Note that in this case no explicit array ordering is provided, in the absence of further information arrays default to row major ordering. In fact we could have omitted all *row* types in the code if we had wished but these are provided to make explicit to the reader how the partitioned data is allocated and viewed.

The assignment $A{:=}S$ at line 14 will result in a scattering of data held in S, which is located on process zero, amongst the processes into each partitioned block of A. In the loop at lines 16 to 19, each process will iterate through the blocks allocated to them and for each block perform the 1D FFT on individual rows. Assignment from A to B at line 21 essentially transposes A and shuffles the blocks of array A across processes. This allows each process to perform linear FFT on the other dimension locally. Because C uses vertical partitioning and is a row major view of the data, performing row-wise FFT on C is the same as performing column-wise FFT on B at lines 23 to 26. The last assignment $S{:=}C$ gathers the data distributed amongst the processes into array S held on process zero.

From the code listing it can be seen that the number of partitioned data blocks is two times the number of processes. Uneven partition sizes, for instance when the number of partitions does not divide evenly into the data size is transparent to the programmer. The types also abstract how and where the data is decomposed and processes can hold any number of blocks with the allocation, communication and transposition all taken care of by the type library. In conventional languages and frameworks it can add considerable complexity when blocks of data are uneven sizes and unevenly distributed, but using the type oriented approach this is all handled automatically. The programmer need not worry about these low level and tricky aspects - unless they want to where additional type information can be used to override the default behaviour.

5.1 Modifying data decomposition and distribution

It is often the case that programmers wish to get their parallel codes working in the first instance and then further tune and specialise if required. Often decisions made early on, such as the method of data decomposition, might not be correct retrospectively but can be very difficult to change without rewriting large portions of the code. Conversely, when orientating the code around types, changing the method of data decomposition is as simple as modifying a type. This will abstract exactly what data is where and allows for the programmer to not only tune but also experiment with different distribution options and how these can affect their code performance and scalability.

In listing 3 the *evendist* type has been used to perform an even cyclical distribution of the data. Instead, the programmer can change one or more of the distribution mechanisms to another distribution type such as array distribution. The *arraydist* type allows the programmer to explicitly specify what blocks reside in the memory of what processes using an integer array. The index of each element in the array corresponds to the block Id and the value held there which process it resides upon. Listing 4 illustrates using array distribution and is a snippet of the Mesham FFT code declaring the distributed arrays. At line 1 the array d is declared to be an array of p integers and in the absence of further information a copy of this is, by default, allocated on all processes. At lines 3 to 5 for every even numbered block Id we are allocating it to process one and uneven block Ids to process two. The arrays A, B and C are then declared to use the *arraydist* type with the array d controlling what blocks belong where. Apart from modifying the type and code for the distribution, all other aspects of the FFT code in listing 3 remain unchanged and the programmer can explicitly change what blocks belong where by modifying the values of the distribution array d.

```
1 var d : array[Int,p];
2 var i;
3 for i from 0 to p - 1 {
4         d[i]:=i % 2 == 0 ? 1 : 2;
5 };
6
7 var A : array[complex,n,n]::allocated[row[]::horizontal[p]::single[
    arraydist[d]]];
8 var B : array[complex,n,n]::allocated[col[]::horizontal[p]::single[
    arraydist[d]]];
9 var C : array[complex,n,n]::allocated[row[]::vertical[p]::single[arraydist
    [d]]]::share[B];
```

Listing 4: Mesham FFT example using array based data distribution

5.2 Results

Whilst the programmability benefits of orienting parallel codes around types have been argued, it is equally important to consider the performance and scalability characteristics of this programming model. We have tested the Mesham version in code listing 3, which uses a CooleyTukey FFT kernel against the Fastest Fourier Transformation in the West version 3 (FFTW3)[5] library. FFTW is a very commonly used and mature FFT calculation framework which looks to optimise the computational aspect of FFT by selecting the most appropriate solver kernel based upon parameters of the data. Performance testing has been carried out on HECToR, the UK National Supercomputer, a Cray XE6 with 32 cores per node, 32GB RAM

per node and interconnection via the Gemini router. Data distribution in both test codes is that of even, cyclical, distribution with one block of data per process. The results presented in this section are the average of three runs.

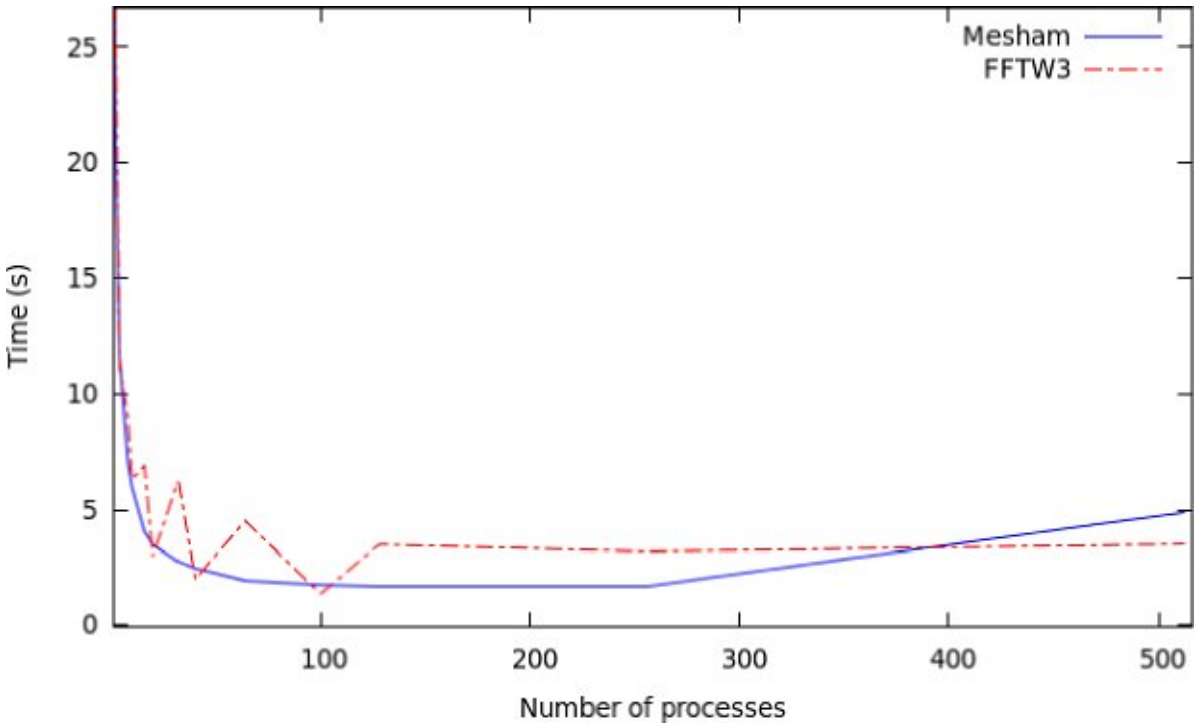


Figure 1: Performance of Mesham FFT version compared to FFTW3

Figure 1 illustrates the performance of the FFT example in Mesham compared with the same problem solved using FFTW3. It can be seen that on small numbers of processes the performance is very similar and both exhibit good scalability as the number of cores is increased initially. There is some instability with the FFTW3 version compared to running the code using an even and uneven partitioning of data. Previous tests using FFTW2 illustrated that that older version of the library performed poorly when run parallel with uneven block sizes of data. Ironically in our tests the latest version, FFTW3, exhibits better performance when run with an uneven partitioning of data compared to an even partitioning. The performance of the Mesham version is more stable and predictable. The rich amount of information available at compile and runtime means that the language is able to select the most appropriate form of communication for specific situations automatically. The *one size fits all* approach of communication adopted by many existing libraries is often optimised for specific cases and does not necessarily perform well in all configurations. At medium numbers of core counts the performance of the Mesham FFT version is more favourable than that of FFTW3 although as we go to larger numbers of processes the Mesham version does degrade faster. Due to the slightly larger overhead of the presently implemented Mesham parallel runtime system, performance degradation sets in somewhat earlier for this strong scaling scenario than in the highly tuned Cray MPI implementation.

Due to the abstractions provided by the PGAS memory model and our use of types, it is entirely possible to maintain correctness of the code whilst running on different architectures although this might have a performance impact. The implementation of Mesham is such that all architecture dependant aspects, for example how specific communications are implemented, are directed through a runtime abstraction layer which can be modified to suit different target machines. The runtime abstraction layer used for the experiments in this paper was for each PGAS processor to be single processes which are connected via MPI. A threading layer also exists which Mesham codes can use unmodified, and an avenue of further work will be to explore how we might optimise performance by selecting or mixing these layers. As previously noted, by changing types the programmer can very easily change key aspects of their code or experiment with different choices such as data decomposition, and this will promote easy tuning to specific architectures. Contrast against more traditional approaches, such as MPI, the porting of these codes to different architectures or mixing paradigms such as OpenMP with MPI often requires substantial and indepth changes to be made.

5.3 Usage in library development

The FFT case study that we have considered in listing 3 simply illustrates the code in a single function. It is worth mentioning the suitability to more advanced codes, or even library development, where data using these complex type representations are passed between functions. In the current implementation of Mesham the entire type chain of a variable must be specified in the formal arguments of a function, which means that the compiler has detailed knowledge of the variables passed to a function and can perform appropriate static analysis and optimisations upon them. At runtime, when passed as an actual argument to a function, data will already have been allocated which occurs as part of a variable's declaration. The Mesham runtime library keeps track of the state of all program variables which means that during execution functions not only know the exact type of data but also its current state. The result is that, for the FFT example, no redistribution of the data would be required if passed to a function.

6 Conclusions

This paper is not intended to describe the entire language Mesham but illustrate the central ideas behind the programming paradigm and demonstrate advantages when applied to the PGAS memory model. Aspects of this paradigm could, in the future, be used as part of existing PGAS languages to get the best of both worlds - a solution which parallel programmers are already familiar with but the added programmability benefits of our approach.

The rationale behind type oriented parallelism is not only to generate a highly efficient parallel executable but also enable programmers to write the source program in an intuitive and abstract style. The compiler essentially helps the programmer determine various sophisticated details of parallelisation as long as such details can be derived from the types in the source program. Optimization algorithms can also benefit from such additional type information. We have used a 2D parallel FFT case study to evaluate the success of our approach, both in terms of programmability with the benefits this affords, and also performance when compared to more traditional solving solutions. It has been seen how the Mesham programmer can architect their code at a high level using language default behaviour and then, by modifying type information, further specialise and tune whereas existing PGAS solutions often impose specific "best effort" decisions upon the programmer. By using types programmers can even

experiment with different choices, such as data decomposition, which traditionally require a much greater effort to modify.

We have compared the performance of the FFT Mesham case study against that of FFTW3. Whereas FFTW3 optimises heavily based upon the computation aspect; our version, where the compiler and runtime optimise the communication based upon the rich amount of type information, performs comparatively and in some instances favourably. There is further work to be done investigating why the performance of the Mesham version decreases more severely than FFTW past the optimal number of processes and we are looking to extend our version to 3D FFT with additional data decompositions such as Pencil. We also believe that Mesham would make a good platform for exploring heterogeneous PGAS, where the complexity of managing data stored on different devices can be abstracted via types. As discussed in section 5.2 all machine dependant aspects are current managed via a runtime abstraction layer, and further development of this could allow for existing codes to be run unmodified on these heterogeneous machines.

References

[1] N. Brown. Mesham language specification, v.1.0. [online], 2013. Available at `http://www.mesham.com/downloads/specification1a6.pdf`.

[2] UPC Consortium. Upc language specifications, v1.2. *Lawrence Berkeley National Lab Tech Report*, LBNL-59208, 2005.

[3] P. Hilfinger et al. Titanium language reference manual. *U.C. Berkeley Tech Report*, UCB/EECS-2005-15, 2005.

[4] G. Luecke and J. Coyle. High performance fortran versus explicit message passing on the isb sp-2. *Technical Report Iowa State University*, 1997.

[5] M.Frigo and S.Johnson. Fftw: An adaptive software architecture for the fft. *IEEE Conference on Acoustics, Speech, and SignalProcessing*, 3:1381–1384, 1998.

[6] R. Numrich and J. Reid. Co-array fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[7] Cray Inc. Seattle. Chapel language specication (version 0.82). [online], October 2011. Available at `http://chapel.cray.com/`.

[8] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123169, 1998.

Additional coarray features in Fortran

John Reid

JKR Associates, UK
Convenor, ISO/IEC Fortran Committee

Abstract

A Technical Specification (TS) is being prepared that extends Fortran 2008 by adding more coarray features. We provide a brief description of the features. Of particular note is the ability to continue execution after the failure of some images.

1 Introduction

Fortran 2008 (ISO/IEC 2010) contains features for parallel programming using a SPMD (Single Program Multiple Data) model. The program is replicated and each replication, known as an *image*, has its own set of data objects. Each image can access its own objects as a traditional Fortran program. In addition, those objects that are declared as *coarrays* may be accessed directly from other images by using an additional set of subscripts, known as `cosubscripts`, in square brackets. For a detailed explanation, see Chapter 19 of Metcalf, Reid, and Cohen (2011).

In February 2008, it was decided to remove some of the coarray features that were in the draft revision of the Fortran standard at that time in order to reduce its size and avoid delays associated with reaching consensus on the detailed design. It was promised that the features that had been removed would be the subject of a Technical Specification (TS) that extended Fortran 2008. This TS was postponed for completion of work on another TS, on further features for interoperability with C. Serious consideration of the TS on further coarray features began early in 2012.

The features that were removed in 2008 were

1. The collective intrinsic subroutines.
2. Teams and features that require teams.
3. The `notify` and `query` statements.
4. Files connected on more than one image, except for the files preconnected to the units specified by `output_unit` and `error_unit`.

The Fortran committee (ISO/IEC JTC1 SC22/WG5) decided in June 2012 that while the overall size and complexity of the features should be similar to that of the features removed in 2008, the details should be different. In particular

1. The collective intrinsic subroutines should be reduced in number, but should include one to provide a general reduction based on a user-written procedure.
2. The team feature should allow a procedure written for all images to execute on a team without any changes.
3. The `notify` and `query` statements should be replaced by support of events.
4. Not to add a special feature for files connected on more than one image.

5. To add support for more atomic operations.
6. To add support for continued execution after the failure of some images.

In the rest of the paper, we will summarize and explain each of these features. For full details, see the current draft (WG5 2013). Note that this is just a draft and the details are likely to change. We will use square brackets `[ ]` to indicate optional syntax items. As far as we know, support of continued operation after the failure of an image has not been attempted in any other language, although similar facilities are being considered for the next MPI specification.

2 Collective subroutines

The following collective subroutines are included. Each invocation must be made by the same statement on all the images of the current team and the invocations must be in the same order. It is to be expected that some synchronization will occur within an invocation but not necessarily at the beginning or end.

`call co_broadcast(source,source_image)` where `source` is a coarray and `source_image` is an integer. This copies the value of `source` on `source_image` to all other images.

`call co_max(source [,result] [,result_image])` where `source` is of a type integer, real, or character. It need not be a coarray. This subroutine computes the maximum value of `source` on all images elementally (as a scalar if `source` is scalar and an array if `source` is an array). If `result` is present, the result is placed there and `source` is not changed; otherwise, `source` is overwritten by the result. If `result_image` is present, the result is defined on image `result_image` and is undefined on other images.

`call co_min(source [,result] [,result_image])` is similar to `co_max` but returns minimum values.

`call co_sum(source [,result] [,result_image])` is similar to `co_max` but sums the elements. It is for types integer, real, and complex.

`call co_reduce(source, operator [,result] [,result_image])` is similar to `co_sum` but is available for any type and applies the user's operator instead of summation. `operator` is a pure elemental function with two arguments of the same type and type parameters as `source` and result of the same type and type parameters. It is required to be mathematically commutative.

All these collectives have optional arguments `stat` and `errmsg` to indicate whether the call was successful and provide error messages.

3 Teams

Teams are defined by scalar values of the derived type `team_type`, defined in the intrinsic module `ISO_Fortran_env`. At any one time, each image executes as a member of a team that is known as the *current team*. Initially, the current team consists of all images. The value of a team variable is formed by executing the statement

```
form subteam(id, team [,new_index=ni] [,stat=st] [,errmsg=em])
```

where *id* is an integer with a positive value and *team* is a variable of type `team_type`. The same statement must be executed on all images of the current team (except those that have failed, see Section 6) and each joins the subteam consisting of all those with the same subteam identifer *id*. For example, the statement

```
form subteam(2-mod(this_image(),2), odd_even)
```

divides the current team into two halves, according to whether the image index is odd or even. If `new_index=` is present, *ni* is an integer that specifies the image index that the executing image will have in the subteam. The values on images of the subteam must all be different and in the range 1 to the number of images in the subteam. The execution of a `form subteam` statement involves synchronization of all the images of the current team. This allows implementations to hold data within the *team* variable to facilitate efficient execution as a team.

The team is changed by execution of the change team construct:

```
change team(team [,stat=st] [,errmsg=emsg])
    block
end team
```

Within *block*, each image executes as part of its subteam, defined by *team*.

Figure 2 at the end of Section 6, which illustrates continued execution with failed images, shows the formation of subteams and the use of the `change team` construct.

There is an implicit synchronization of all images of the subteam of the execting image and another implicit synchronization of these images at the `end team` statement. The team that was current when the `change team` statement was executed is known as the *parent team* of the team that executes *block*. This parent-child relationship gives any executing team a chain of ancestor teams. The *team distance* from a team to an ancestor is the number of links in this chain. The intrinsic function

```
team_depth()
```

returns the team distance to the original team of all images.

The intrinsic function

```
subteam_id([distance])
```

returns the subteam identifier that was used in the `form subteam` statement that constructed the current team (`distance` absent) or the ancestor team at team distance `distance`.

An optional argument `distance` has been added to the intrinsic function `this_image` to return the image index of the executing image when it was part of the ancestor team at team distance `distance`.

An optional argument `distance` has been added to the intrinsic function `num_images` to return the number of images in the ancestor team at team distance `distance`.

To preserve the ability of implementations to use symmetric memory, where each coarray is stored from the same location on each image, any allocatable coarray that was allocated before the change team block is executed must not be deallocated during its execution and any allocatable coarray that is allocated during its execution is automatically deallocated on its

completion unless it has already been deallocated.

All cosubscripts and image indices are normally relative to the current team, but syntax has been added to for cosubscripts to refer to an ancestor team. For example

```
a[ancestor:: i,j]
```

refers to the coarray `a` on the image for which it has cosubscripts `i,j` within the team `ancestor`. This makes data on images outside the current team accessible. All the synchronization statements of Fortran 2008 apply only to the images of the current team, but the images of any team may be synchronized by execution of the statement

`sync team(`*team* `[,stat=`*st*`] [,errmsg=`*emsg*`])`

including a team that is an ancestor, a descendant, or neither.

4 Events

An image can use an `event post` statement to notify another image that it can proceed to work on tasks that use common resources. An image can wait on events posted by other images and can query if images have posted events.

Events are stored in scalar coarray variables of the derived type `event_type` defined in the intrinsic module `ISO_Fortran_env`. Each holds a count of the difference between the number of successful posts and successful waits. The initial value of this count is zero.

Events are posted by executing the statement

`event post(`*event* `[,stat=`*st*`] [,errmsg=`*emsg*`])`

and waits occur by execution of the statement

`event wait(`*event* `[,stat=`*st*`] [,errmsg=`*emsg*`])`

If the count of the event is zero, a wait occurs until it becomes positive. A successful wait decrements the count by one. The event in an `event wait` statement is not permitted to be coindexed, that is, it must be a local variable. This restriction allows efficient execution of the `event wait` statement.

There are restrictions to prevent the value of an event changing other than through the execution of `event post` and `event wait`. Events may be queried with the intrinsic subroutine

```
call event_query(event, count [,status])
```

which provides the current event count for `event`. The integer `status` is given the value zero after a successful query and a nonzero value otherwise.

An example involving a producer-consumer program is shown in Figure 1.

5 Atomic operations

In Fortran 2008, a sequence of statements executed on an image between two image control statements such as `sync all` is known as a *segment*. In general, the programmer is required to ensure that if a variable is defined in a segment it is not referenced on defined in a segment on

```
use, intrinsic :: iso_fortran_env
integer :: i, count
type(event_type) :: event[*]
i = this_image()
do
   i = i + 1
   if (i>num_images()) i = 1
   if (i/=this_image()) then
      call event_query(event[i], count)
      if (count==0) then
         ! Produce work packet for image i
         event post(event[i])
      end if
   else
      event wait(event)
      ! Consume work packet from another image
   end if
   ! If all work completed, exit
end do
```

Figure 1: Producer-consumer program.

another image unless the segments are ordered, say by the use of the `sync all` statement.

However, this rule may be broken for a scalar variable `atom` of type `integer(atomic_int_kind)` or `logical(atomic_logical_kind)`, whose kind value is defined in the intrinsic module `iso_fortran_env` by calling an *atomic* subroutine. The effect of executing an atomic subroutine is as if the action on the argument `atom` occurs instantaneously and thus does not overlap with other atomic actions that might occur asynchronously. The allowed types are limited because it is anticipated that there may be special hardware to support atomic actions on them.

Fortran 2008 contains only two atomic subroutines, `atomic_define` and `atomic_ref`. The TS adds five more atomic subroutines.

For the atomic addition of two integers, the intrinsic subroutine

```
call atomic_add(atom, value [,old])
```

is provided. Here, `atom` is a scalar of type `integer` with kind `atomic_int_kind` and `value` is a scalar integer. It is overwritten atomically by adding `value` to it. If `old` is present, it is given the previous value of `atom`.

For bitwise operations on the bit in two integers, the intrinsic subroutines

```
call atomic_and(atom, value [,old])
call atomic_or(atom, value [,old])
call atomic_xor(atom, value [,old])
```

are provided. The arguments are as for `atomic_add` but apply the specified bit operation.

For atomic compare and swap for two integers or logicals, the intrinsic subroutine

```
call atomic_cas(atom, old, compare, new)
```

is provided. Here, `atom` is a scalar of type `integer` with kind `atomic_int_kind` or of type `logical` with kind `atomic_logical_kind`, `old` is of the same type as `atom`, `compare` is of the same type and kind as `atom`, and `new` is of the same type as `atom`. If `atom` and `compare` have the same value, `atom` is given the value of `new`. The variable `old` is given the previous value of `atom`.

6 Failed images

A failed image is one for which references or definitions of variables fail when that variable should be accessible, or the image fails to respond as part of a collective activity. A failed image remains failed for the remainder of the program execution.

The value `stat_failed_image` in the intrinsic module `ISO_Fortran_env` is a possible return value for a `stat=` specifier for any of these statements `change team`, `end team`, `form subteam`, `sync all`, `sync images`, `sync memory`, `sync team`, `lock`, `unlock`, `event post`, `event wait`, `allocate`, or `deallocate`, and for the `stat` argument of a collective subroutine. This return value will occur if there is a failed image in the current team. In the case of `sync all`, `sync images`, or `sync team`, the statement will have successfully synchronized all the images of the specified set that have not failed. In the case of `form subteam`, the statement will have successfully formed a subteam of those images that have not failed. If an `allocate` or `deallocate` statement is otherwise successful, the allocations and deallocations will have been performed on the images that have not failed and these images will have been synchronized. For `lock`, `unlock`, `event post`, `event wait`, or a collective subroutine, the desired effect will probably not have been achieved. If no status option is specified for any of the statements mentioned in this paragraph and a failed image is involved in its execution, the program is terminated.

The intrinsic function `num_images` has been extended to have an optional argument `failed` as well as the additional optional argument `distance` (see Section 3). If `failed` is present with the value true, the number of failed images in the current team, or the ancestor team at team distance `distance`, is returned and if `failed` is present with the value false, the number of such images that have not failed is returned.

Which images have failed may be determined by the function

```
failed_images([kind])
```

This returns an integer array holding the image indices of the failed images in the current team in increasing order. If `kind` is present, it specifies the kind parameter of the result.

Figure 2 shows how a code that can continue after a failure might be written. The images are divided into two halves that each redundantly perform the whole required calculation. Periodically, each checks to see if the corresponding image of the other half has failed. If it has, it checks its own team and if this too has a failure, terminates the whole calculation.

```
use ISO_Fortran_env ! Specifies team_type and stat_failed_image
integer :: i,me,p,st
type(team_type) :: half,partner
me = this_image()
p = num_images()
i = 1; if (i>p/2) i = 2
form subteam(i, half)
i = me; if (i>p/2) i = me-p/2
form subteam(i, partner)
change team (half)
    ! Perform whole calculation
        :
    sync team(partner,stat = st)
    if (st==stat_failed_image) then
    ! Partner has failed. Check my team.
       sync memory(stat=st)
       if (st==stat_failed_image) error stop
    end if
      :
end team
```

Figure 2: Contining after an image faliure.

Another possibility for recovery is for the executing images to form a new team for continued execution:

```
if (num_images(failed=.true.) > 0 ) then
   form subteam(1, recover)
   sync all (stat=st) ! Will return stat_failed_image
   change team (recover)
       : ! Execute as a subteam
   end team
end if
```

7 Acknowledgement

I would like to express my appreciation of the help of Bill Long of Cray Inc., who has read drafts of this paper and suggested many corrections and improvements.

8 References

ISO/IEC (2010). ISO/IEC 1539-1:2010 Information technology – Programming languages – Fortran – Part 1: Base language. ISO, Geneva.

WG5 (2013). ISO/IEC SC22/WG5/N1983. Draft TS 18508 Additional Parallel Features in Fortran. Available from `http://www.nag.co.uk/sc22wg5/`.

Metcalf, M,. Reid, J., and Cohen, M. (2011). Modern Fortran explained. OUP.

Guiding X10 Programmers to Improve Runtime Performance

Jeeva Paudel[1], Olivier Tardieu[2] and José Nelson Amaral[3]

[1] Computing Sc., University of Alberta, Edmonton, Alberta, Canada
jeeva@cs.ualberta.ca
[2] IBM T. J. Watson Research Center, Yorktown Heights, NY, U.S.A.
tardieu@us.ibm.com
[3] Computing Sc., University of Alberta, Edmonton, Alberta, Canada
amaral@cs.ualberta.ca

Abstract

Performance modeling and tuning parallel and distributed applications often require comprehension of a large and complex code base to identify static locations in the source code and dynamic points in the program execution that are performance hotspots or potential pitfalls. Proper tool support is essential for these tasks. This paper describes *XAnalyzer*, a first-of-its-kind framework that uses pointcuts-and-advice aspects to identify patterns in the source code and program execution structure that are possible causes of performance bottlenecks, or that may be restructured for better performance. XAnalyzer also offers suggestions to remedy the problems and improve runtime performance. Evaluation of XAnalyzer on a number of benchmarks illustrates its correctness. Also, the performance improvements, in the range of 7% and 87%, achieved through program restructuring based on the XAnalyzer's report illustrates its importance and usefulness.

1 Introduction

An understanding of the performance model of a language helps programmers develop efficient programs. Without proper tool support, programmers must remember critical implementation details of several high-level language constructs, and must also navigate and comprehend large pieces of code to identify source locations that may benefit from performance tuning. For concurrent and distributed programming paradigms, such as Asynchronous Partitioned Global Address Space languages [6], identifying such locations is even more challenging.

This paper presents *XAnalyzer*, a framework that identifies patterns in source code and execution behaviour of an X10 application. These patterns may be potential sources of performance bugs and the application may benefit from the use of alternative language constructs or simple code refactoring. XAnalyzer offers suggestions that programmers may use to alleviate the bottlenecks or to improve performance. The overarching goal of XAnalyzer is to reduce the cognitive effort and the time required to diagnose and tune performance-critical pieces of code and to enhance reasoning about the performance model of an application.

A common way to identify specific source-code patterns in a program is using static analysis. For example, Vasudevan *et al.* [10] combine clock analysis and aliasing analysis to identify patterns of clocks (dynamic barriers) in X10 programs that do not require full-blown implementation of a typical clock. The default clock implementation in X10 enables distributed activities to synchronize. However, if all activities registered to a clock belong to the same place, a much faster clock implementation that avoids the overhead of a distributed synchronization protocol is possible.

Contrary to mainstream approaches for pattern identification, XAnalyzer uses aspect-orientation and also relies on program execution behaviour because: (i) it is a unified performance-diagnostic framework that encompasses both statically identifiable patterns and also the ones that manifest during runtime; (ii) *pointcuts* in aspects offer a concise mechanism to identify and describe the patterns of interest in both source-code locations and program-execution points; (iii) *aspect-orientation* provides a modular and pluggable implementation with the guarantee of a zero cost of instrumentation when turned off. It is difficult to achieve a similar guarantee when compiler and runtime implementations are instrumented.

XAnalyzer correctly identified important source-code locations, constructs and optimization opportunities that are critical to performance in several applications used in its evaluation. The performance benefits achieved in applications tuned with such information demonstrate that the insights provided by XAnalyzer are important and useful.

2 Related Work

There are several tools, and communication interfaces that characterize parallel applications for high-performance tuning on multicore clusters. However, they are neither tailored for the Asynchronous Partitioned Global Address Space (APGAS) programming model nor they cater to the unique needs of the X10 programming language owing to its semantics and implementation.

The PerfExpert [2] tool detects core, socket, and node-level performance bottlenecks inside loops and procedures. It uses a local-cost-per-instruction metric to identify and report performance pitfalls. The scope of XAnalyzer is comparatively broader – it analyzes parallel constructs such as async, place, and the communication structure between places, besides loops and procedures, to identify possible characterizations of interest to the programmer.

Diamond *et al.* [3] use hardware metrics such as cache miss rates, shared off-chip memory bandwidth, and DRAM page conflicts to determine the scalability of applications on multicore chips and multi-chip nodes. While their analyses can complement XAnalyzer, their work differs from ours in two ways. First, they derive scalability information from an application's execution characteristics, while XAnalyzer focuses both on source code constructs, and dynamic execution behaviour to diagnose possible performance pitfalls – including scalability concerns, such as excessive remote memory accesses. Second, XAnalyzer goes further and provides possible suggestions as well to tune the applications.

The Global Address Space Performance (GASP) interface [9] is a standard framework for performance analysis of GAS languages and libraries. GASP assumes that most PGAS languages use GASnet [1] for communication. However, HPC compilers, such as X10, are not strictly using the same interface. Therefore, GASP is not applicable to all of them. Also, coupling compiler and instrumentation code complicates future modification of the performance monitoring code. XAnalyzer is completely decoupled from the compiler and the runtime code.

The Parallel Performance Wizard (PPW) [8] identifies performance bottlenecks and their causes in PGAS applications. Based on critical-path, scalability, and load-balancing analyses, PPW also provides suggestions for removing bottlenecks. Unfortunately, PPW is tailored for PGAS SPMD languages such as UPC and Co-Array Fortran, and does not support X10. Further, PPW's analyses may not be meaningful in the context of X10. For example, visualization of array distribution is meaningful in UPC because the visualization shows how each statically-allocated shared array is distributed across the threads at runtime. In X10, the portion of a distributed array in a place is equally visible and directly accessible to all the place-local threads. Further, X10's Asynchronous PGAS semantics poses significant challenge to directly use PPW – X10 is different from these languages and uses different semantic constructs, code

generation techniques, and compiler and runtime implementations.

X10DT [11], X10's integrated development tool, offers development environment, but does not offer suggestions to programmers on potential source-code locations and constructs that may be sources of performance bottlenecks.

In the absence of a targeted performance diagnosis and reporting tool and given the difficulty in porting other available tools to fit X10's needs, XAnalyzer provides a modular and pluggable implementation to analyze X10 programs with avenues for future extensions.

3 X10 Preliminaries

Every computation in X10 is an asynchronous activity. An X10 place is a repository for related data and activities. Every activity runs in a place. X10 provides the statement `async (p) S` to create a new activity at place p to execute S. Places induce the notion of locality. The activities running in a place may access data located at that place with the efficiency of on-chip access. Access to a remote place may take orders of magnitude longer, and is performed using the `at (p) S` statement. An `at` statement shifts the control of execution of the current activity from the current place to place `p`, copies any data that is required by the statements `S` to `p`, and, at the end, returns the control of execution to the original place. The necessary data copying is done through the runtime system calls inserted by the compiler.

X10 provides `DistArrays`, *distributed arrays*, to spread data across places. An underlying `Dist` object provides the distribution, telling which elements of the `DistArray` go in which place. `Dist` uses subsidiary `Region` objects to abstract over the shape and even the dimensionality of arrays. Specialized X10 control statements, such as `ateach`, provide efficient parallel iteration over distributed arrays [7].

X10 provides cross-place references to a shared variable using `GlobalRef[T]`. X10 also supports defining objects with distributed state using `PlaceLocalHandle[T]`.

4 Design Overview

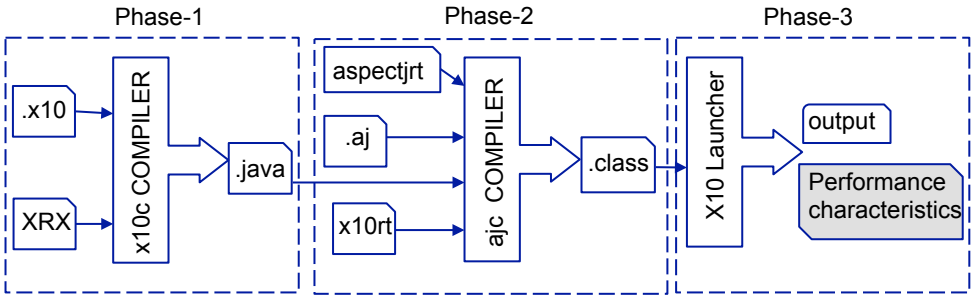


Figure 1: Architecture of XAnalyzer.

The architecture of XAnalyzer is shown in Figure 1. For lack of an aspect-orientation language for X10, XAnalyzer first compiles the X10 user program to Java, and weaves the pointcuts-and-advice aspects that encode the rules identifying performance-critical patterns of interest into the Java code. The aspect-woven byte-code then executes and reports a concise assessment of the program. Such a design of aspects based on the internals of the X10 compiler

offers a direct linkage between the runtime behaviour and the contributing static structures. Further, the design also offers XAnalyzer additional information about programs for performance diagnosis that may not always be available or evident from the X10 source code. For example, it is not apparent from the X10 source code if an `at (p) S` statement is actually serialized or not, but the Java intermediate code explicitly exposes Java's serialization calls to reflect this information. The demerit of relying on the Java intermediate code is that the same compilation scheme may not be adopted by other APGAS compilers.

The design of XAnalyzer provides a modular implementation that does not tangle or cut across the X10 compiler or the runtime. As a stand-alone pluggable implementation, it can be modified or extended separately and without any changes to other parts of the code.

5 Performance-critical Code Patterns

We explored a wide-body of patterns in X10 programs, but XAnalyzer encodes only those patterns that are critical to performance. They are reported in Table 1. XAnalyzer's suggestions about possible code refactoring are not based on proven program-analysis techniques. Thus, the suggestions are not necessarily precise or sound, and users must use them only as guidelines per se. Nevertheless, information about possible opportunities for optimization at specific source locations, or language constructs is valuable during performance tuning.

5.1 False Place Shifts

To perform a place shift, the X10 code generator creates a closure that executes statement `S`, serializes the closure, sends the serialized data to place `p`, and executes the closure in place `p`. The closure encapsulates an isomorphic copy of the object graph reachable from `S`. The caller of the `at` statement is blocked until the `at` statement returns. These operations are expensive but necessary for shifting part of an activity to a remote place. However, for false place-shifts, where an activity is shifted to its current place, such operations are unnecessary overheads and must be avoided. The X10 implementation may be able to eliminate or otherwise optimize some of this serialization, but it must ensure that any program visible side-effects caused by user-defined custom serialization routines happen just as they would have in an unoptimized program [4]. Thus, automatic optimization of serialization requires a complex global analysis to ensure that it does not interfere with user-defined serializations. Furthermore, performing global analysis to identify and preclude such false place-shifts may incur high overheads that may not be offset by the resulting benefits. Therefore, even for a local place (*i.e.*, `p == here`) execution of `at` statement entails copying and serialization of all the program state referenced by `S`. However, such overheads can be easily avoided by simple code re-structuring:

```
at (p) S  ⟹  if (p == here)
                S;
              else
                at (p) S;
```

For the example in Figure 2, XAnalyzer identifies the source code locations that involve false place-shifts and reports them as follows [1]:

line 117: KMeansDist.x10: False place-shift `at (there) atomic {`

[1] XAnalyzer reports all occurrences of such place shifts. For brevity, we show only occurrence of each program pattern in this section.

```
110 finish {
111   val clustrsGr = GlobalRef(clustrs);
112   val numClustrsGr = GlobalRef(numClustrs);
113   val there = here;
114   for(d in points_dist.places()) async {
115     val tnewClustrs = lnewClustrs();
116     val tClustrCounts = lClustrCounts();
117     at(there) atomic {
118       for(var j:Int=0; j<DIM*CLSTRS; ++j) {
119         clustrsGr()(j) += tnewClustrs(j);
120       }
121       for(var j:Int=0; j<CLSTRS; ++j) {
122         numClustrsGr()(j) += tClustrCounts(j);
123       }
124     }
125   }
126 }
```

Figure 2: Example of a false place shift.

5.2 Distributed Asyncs and Finish

X10 permits arbitrary patterns of task creation and termination through flexible combination and nestings of `at` and `async` statements inside `finish`. If a `finish` encloses remote `async`s, detecting their termination entails communication costs and latency. Each place maintains a counter for each finish statement to keep track of the activities spawned and terminated. Once all activities in a place terminate, the local quiescence event is transmitted to the `finish` place. These costs of distributed termination detection increase with the number of places involved in finish. Therefore, the X10 runtime dynamically optimizes `finish` by optimistically treating as if it encloses only local `async`s and then dynamically switching to a more expensive distributed algorithm only upon encountering an `at` inside `finish`. The runtime also performs sparse message encodings, message coalescing, and lazy allocation of counters to minimize these overheads.

The runtime also provides efficient implementations specialized for commonly occurring patterns of distributed concurrency. For example, X10 provides the FINISH_DENSE pragma to indicate that a `finish` does not need to monitor spawning or termination of `async`s in remote places. Consider an example where a remotely spawned child `async` spawns an `async` back at its parent's place. Such a `finish` is optimized by ignoring both the termination of the parent task in the current place and the spawning of the child task in the remote place. The `finish` only needs to track `async` creation at the local place with the termination of the child task at the remote place. X10 also provides optimized implementation of five other forms of `finish`, but relies on user guidance to identify the particular pattern of `async`s enclosed by the `finish`.

In the absence of a fully automated compiler analysis that can identify and apply the related optimizations, the case for XAnalyzer becomes even more stronger. An overview of the places involved in a `finish`, and also its nesting structure helps programmers make better use one of these available pragmas. XAnalyzer reports distributed finish statements as:

line XXX: finish {place(0), place(1), place(2), place(3)}

The line number indicates the location of `finish` in the program.

5.3 Asynchronous Execution

An activity is created by executing the async statement: `async S`, which starts a new activity located here executing S. To start a new activity at a specific place, X10 offers the `at (p) async S` construct. The costs of executing `async`, `at (here) async` (*i.e.*, p==here) and `at (p) async` are different. The `async` construct works directly on the place-resident data and does not need to create their copies. However, the execution of `at (here) async` creates copies of all the values used in `S` to the local place, *i.e.*, `here`, even though there is no actual change of place, and the original values already exist in the place. Thus, the cost of running `at (here) async` is higher than that of running `async`. Further, the cost of running `at (p) async` is higher than that of running `at (here) async` because it involves inter-node communication to create the copies at the remote node `p`. Therefore, it is important to use the appropriate async construct for local and remote places.

However, the overheads of data copying, closure creation and serialization can be easily avoided by a simple code re-structuring as shown below:

```
async at (p) S  ⟹  if (p == here)
                     async S;
                   else
                     at (p) async S;
```

For the FSSimpleDist program used in our evaluation, XAnalyzer reports such source code locations as:

```
line 46: FSSimpleDist.x10: Async serialized locally
async at (Place.place(p)) {
```

5.4 at (p) async vs async at(p)

For remote execution of tasks, X10 offers two possible ways of combining `async` and `at` statements: `at (p) async` and `async at (p)`. The first form evaluates the expression `p` synchronously using the parent `async` activity, while the second form evaluates `p` asynchronously with the parent `async` as it is enclosed in the body of the `async`. In many cases, using any of these two forms interchangeably is semantics preserving.

When the expression `p` can be evaluated statically, `at (p) async` is preferred over `async at(p)`. The execution of `at (p) async` is translated into a single asynchronous message to the remote place `p`, while the execution of `async at (p)` creates an activity at the current place and shifts that activity to the remote place. Shifting the place of an activity in this manner is significantly more expensive than the former form. XAnalyzer alerts programmers about such opportunities for using cheaper forms of `async` as:

```
Found async at (p): at (p) async is preferred
```

5.5 Excessive Serialization

A large number of place shifts incurs communication and serialization overheads. Frequent place shifts may occur if data is declared local to a place but accessed by activities from several other places. A high-level characterization of place shifts in an application may encourage programmers to test other constructs such as `GlobalRef`s, where they have greater control over the data that is serialized. XAnalyzer summarizes the cross-place "at" references as:

Cross-place “at” references:

{place(1) → place(0), place(2) → place(0), ...}

where `place(1)` → `place(0)` represents a place-shift from `place` 1 to `place` 0.

5.6 Expensive Data Capture

The execution of `at` may entail copying of significantly more data than programmers may realize. For example, accessing a field of an instance or calling an instance method will require capturing the entire object using `this`. One way to avoid capturing expensive data is to manually refactor the code to extract the parts of an object that are actually referenced within the `at` statement into local variables and then to use them (instead of the entire object) in the body of the `at` statement. For the example in Figure 3, XAnalyzer identifies and reports such

```
1  public class FieldRefInAt {
2    static class Foo {
3      private val selfRef = GlobalRef[Foo](this);
4      public var flag: boolean;
5      // transient public var flag: boolean;
6    }
7    public def setField() {
8      val nxtPlace = here.next();
9      val baz = (at(nxtPlace) new Foo()).selfRef;
10     at (baz) { baz().flag = true; }
11     return true;
12   }
13   public static def main(Array[String](1)) {
14     val res = new FieldRefInAt().setField();
15   }
16 }
```

Figure 3: Example of field access inside `at` statement.

expensive data capture as:

line 10: FieldRefInAt.x10: Expensive “this” capture inside “at”

For this synthetic example, capturing the entire object with `this` can be avoided by qualifying the flag variable as `transient`, as shown in line 5 in Figure 3.

5.7 For-loop index

In X10, `for`-loop optimization is highly sensitive to trivial details of how the loop is written. The X10 compiler converts some `for` loops over rectangular regions into counted `for` loop nests. The optimizations apply only on `for` loops that iterate over a region or an array using an un-named exploded-form index [2]. The snippet of code in Figure 4 shows a `for` loop that uses the name `p` to iterate over the `intArray` (line 7).

X10 currently does not optimize such loops. XAnalyzer identifies such loops and alerts the user as follows [3]:

[2]For a 2D array `r` of size 1..10*1..10, an example of a `for` loop with un-named exploded-form of index is: `for([i,j] in r)`.

[3]The AspectJ compiler does not provide pointcuts to capture joinpoints on loops. Therefore, we use the AspectBench compiler to identify for-loops.

```
1  public class UnOptimizedForLoop {
2    public static def main(Array[String](1)) {
3      val r = 1..5*1..5;
4      val intArray = new Array[Int]
5                     (r, ([i,j]:Point(2))=>i*j);
6      //for ([i,j] in intArray) {
7      for (p in intArray) {
8        intArray(p) += intArray(p);
9        //intArray(i,j) += intArray(i,j);
10     }}
11 }
```

Figure 4: Example of for loop that will not be optimized.

line 7: UnOptimizedForLoop.x10: Unexploded named point found in for-loop: Exploded un-named form of index preferred

Simple refactoring of the `for` loop to use un-named exploded form of index, as shown by the commented lines in Figure 4, will allow the compiler to optimize the loop.

5.8 Struct vs Class

In X10, an object implementation requires some extra storage to save its runtime class information and also a pointer for each reference to the object. An object also requires an extra level of indirection to read and write data, and also some runtime computation for dynamic dispatch. Such space and time overheads can be prohibitive in high performance computing if incurred on all objects. X10 provides *structs*, stripped-down objects, to mitigate these overheads. Structs are less powerful than objects because they lack inheritance and mutable fields. Without inheritance, method calls do not need to do any lookup and can be implemented directly. Thus, structs avoid some space and time overhead that objects typically incur.

`Struct` may not always be a preferred choice for performance tuning over `class`. For instance, using an array of `struct` can offer higher locality benefits over using an array of references to objects (owing to random heap locations where objects are allocated). However, if an application uses several references to objects, creating multiple instances of structs would be significantly more expensive than using objects. Further, refactoring a `class` to a `struct` may require further semantics-preserving changes to other parts of the code. `instanceOf` relationship, for example, would no longer apply to structs.

Thus, a programmer must eventually decide whether restructuring a `class` as a `struct` leads to performance gains, and preserves the correctness of the entire application, or necessitates some code changes. But having a choice about possible optimization opportunities is nevertheless a useful information. XAnalyzer informs programmers about such an opportunity as follows:

No inheritance and “var” fields in Class: Struct is preferred

Figure 5 shows a snippet of code from the GCSpheres benchmark. The `Vector3` class inside the GCSpheres class does not inherit any other class or interface and does not declare any field as `var`. Such a class can be easily refactored as a `struct` as shown by the comment in line 3.

```
1  class GCSpheres {
2    static type Real = Float;
3    //static struct Vector3(x:Real,y:Real,z:Real){
4    static class Vector3(x:Real,y:Real,z:Real){
5      public def getX () = x;
6      public def getY () = y;
7      public def getZ () = z;
8      public def add (other:Vector3)
9        = new Vector3(this.x+other.x,
10                     this.y+other.y,
11                     this.z+other.z);
12     public def neg ()
13       = new Vector3(-this.x, -this.y, -this.z);
14     public def sub (other:Vector3)
15       = add(other.neg());
16     public def length () = Math.sqrt(length2());
17     public def length2 () = x*x + y*y + z*z;
18   }
19   ...
20 }
```

Figure 5: Example of Class that can be refactored as Struct.

6 Use of Aspects in XAnalyzer

A detailed description of Aspect Oriented Programming (AOP) is beyond the scope of this paper. AOP is introduced in a seminal paper by Kiczales *et al.* [5]. This section provides a high level overview of one aspect used in XAnalyzer to illustrate the use of aspects in XAnalyzer.

Figure 6 shows a snippet of aspect that identifies and reports `async` statements that are serialized to the local place. The X10 compiler internally converts `async at(p) S` into `runAsyncAt(..)` calls. The pointcut (line 16) `runAsyncAtCall()` captures all calls to the `runAsyncAt(..)` method in the user program. The `before()` advice (line 18) captures the `place` where the async statement is executed and stores it in `placeId` (line 19). If the place is local, the advice retrieves and stores the line number of the `runAsyncAt(..)` call. Then, after the main method of the user program executes (line 4), the advice prints the source locations that lead to serialization of local `asyncs`, and also prints out the first line of code related to the `runAsyncAt(..)` method.

7 XAnalyzer Evaluation

This section describes the benchmarks, experimental platform and experiments used in the evaluation of XAnalyzer.

7.1 Benchmarks

The benchmarks used in the experimental evaluation were chosen from the standard X10 compiler release, except for the ForLoop micro-benchmark which we wrote ourselves.

(a) ForLoop is a synthetic benchmark that performs element-wise copy of an Integer array of size 1000.

(b) NQueensDist is a distributed NQueens problem.

```
public aspect LocalAsyncSerializationReporter {
  pointcut mainMethod():
    execution(public static void main(..));
  after() returning: mainMethod() {
    Iterator it = locSetComments.iterator();
    while(it.hasNext()) {
      int line=((Integer)it.next()).intValue();
      showLines(filename, line);
      extractLineNums(filename, line);
    }
 }
 static int commentLine = 0;
 Set<Integer> locSetComments =
   new HashSet<Integer>();
 pointcut runAsyncAtCall():
   call (* runAsyncAt(..));
 before(): runAsyncAtCall() {
   Place placeId
       = (Place)thisJoinPoint.getArgs()[0];
   if (placeId.id == 0){
     commentLine
         = tjp.getSourceLocation().getLine();
     codeLoc = new Integer(commentLine-2);
     locSetComments.add(codeLoc);
    }
  }
}
```

Figure 6: Aspect that diagnoses serialization of local async.

(c) FRASimpleDist is a simple version of the HPC RandomAccess benchmark.

(d) FSSimpleDist is a simple version of the HPC Stream benchmark.

(e) GCSpheres represents the real-world problem in graphics engines of determining which objects in a large sprawling world are close enough to the camera to be considered for rendering.

(f) StructSpheres is an optimized version of GCSpheres that uses struct instead of a static class.

(g) KMeansDist clusters 2000 points into four clusters.

(h) MontePi computes the value of π using the Monte-Pi algorithm.

(i) Histogram computes the histogram of an array A into Rail B.

(j) HeatTransfer solves 2D partial differential equations that can be expressed as iterative 4-point stencil operations.

(k) SSCA1 is the HPC Scalable Synthetic Compact Application that provides analytical schemes to identify similarities between sequences of symbols.

7.2 Experimental Setup

All performance measurements were done on a blade server with 8 nodes, each featuring two 2 GHz Quad-Core AMD Opetron processors (model 2350), with 8 GB or RAM and 20GB

swap space, running CentOS GNU/Linux version 6. One worker thread was used in each core, amounting to a total of 64 threads. Each benchmark was run ten times, and the bars in the performance graphs report their average along with the 95% confidence intervals.

7.3 Correctness and Applicability

Used on a number of benchmarks, XAnalyzer provided a concise and correct assessment of program characteristics related to performance. XAnalyzer suggested alternative constructs to remedy performance pitfalls and code refactorings to improve program performance. Table 1 shows the list of program characterizations reported by XAnalyzer. XAnalyzer also displays the relevant pieces of code, and the source-code locations for each program characteristic as discussed in Section 5. For space reasons, they are not shown in Table 1. The table also describes (in italics) information that is suppressed by XAnalyzer because it is either expected characteristic or is not relevant to performance tuning and modeling.

For each benchmark, the program patterns identified and the corresponding reports generated by XAnalyzer are shown in Table 2.

	Program Patterns Critical to Performance
A	Unexploded named point found in `for` loop: Exploded un-named form of index preferred
B	Expensive `this` capture inside `at`: Extract values needed inside `at` into local variables
C	Cross-place at references: $\{p_1 \rightarrow p_0, ..., p_n \rightarrow p_0\}$
D	*`ateach` characterization not yet supported*
E	Distributed Finish $\{p_1, .., p_n\}$
F	Found `async at (p)`: `at (p) async` is preferred
G	`async` serialized locally: Use `async S (if p == here)`
H	`GlobalRef` references: $\{p_1 \rightarrow p_0, p_2 \rightarrow p_0, ...\}$: `PlaceLocalHandle` may be an alternative
I	No inheritance and `var` fields in `Class`: `Struct` is preferred
J	Found false place-shift: Use `S (if p == here)` instead of `at (p) S`

Table 1: Program patterns identifiable by XAnalyzer.

7.4 Performance Benefits of Optimization

Figure 7 shows the execution time of the benchmarks – the *Opt* version of each benchmark was obtained by restructuring it using the information displayed by XAnalyzer. The *UnOpt* version

Benchmarks	Program Characterizations									
	A	B	C	D	E	F	G	H	I	J
ForLoop	✔									
NQueensDist		✔	✔	✔						
FRASimpleDist		✔			✔	✔	✔			
FSSimpleDist					✔	✔	✔	✔		
KMeansDist					✔	✔	✔			✔
MontePi			✔		✔					✔
GCSpheres									✔	
Histogram										
HeatTransfer										
SSCA1					✔	✔	✔			

Table 2: Patterns in benchmarks identified by XAnalyzer.

is the unoptimized one.[4]

FRASimpleDist benchmark was optimized by preventing serialization of an `async` to the local place, and also by replacing `async at(p)` with `at(p) async`. The program contains one occurrence of each of these constructs. Thus, the performance benefit resulting from remedying them is small, *i.e,* 7%.

ForLoop micro-benchmark achieved a performance gain of 60% only from restructuring of the `for` loop's index to use un-named exploded form.

KMeansDist achieved the largest performance gain (87%) among the benchmarks studied. KMeansDist is an iterative clustering algorithm. In each of its 50 iterations, the unoptimized KMeansDist clustering algorithm includes: a) three instances of `async` that are spawned locally and serialized to the same place, b) one instance of false place shift, and c) three occurrences of `async at(p)` that could be restructured as `at (p) async`.

The contribution of each optimization removing these performance pitfalls is shown in Figure 8. Performance gains achieved from each individual program restructuring is small and within the 95% confidence interval of one another. However, when combined together, the optimizations show significant gains. The highest gain is achieved by combining all optimizations, as indicated by the last bar in Figure 8.

SSCA1 was optimized using `at(p) async` instead of `async at(p)`, and also by preventing local serialization of `async` statement. The SSCA1 implementation has one occurrence of each of these constructs in the code, but they are executed several times inside a loop, and also over several places. Thus, the resulting performance gain is significant, *i.e.*, 42% over the unoptimized version.

MontePi's unoptimized version uses `PlaceLocalHandle` to store local results at each place, and then gathers the local results from each place at the root place to compute the value of

[4]Only the execution time for benchmarks where meaningful optimizations could be performed is shown.

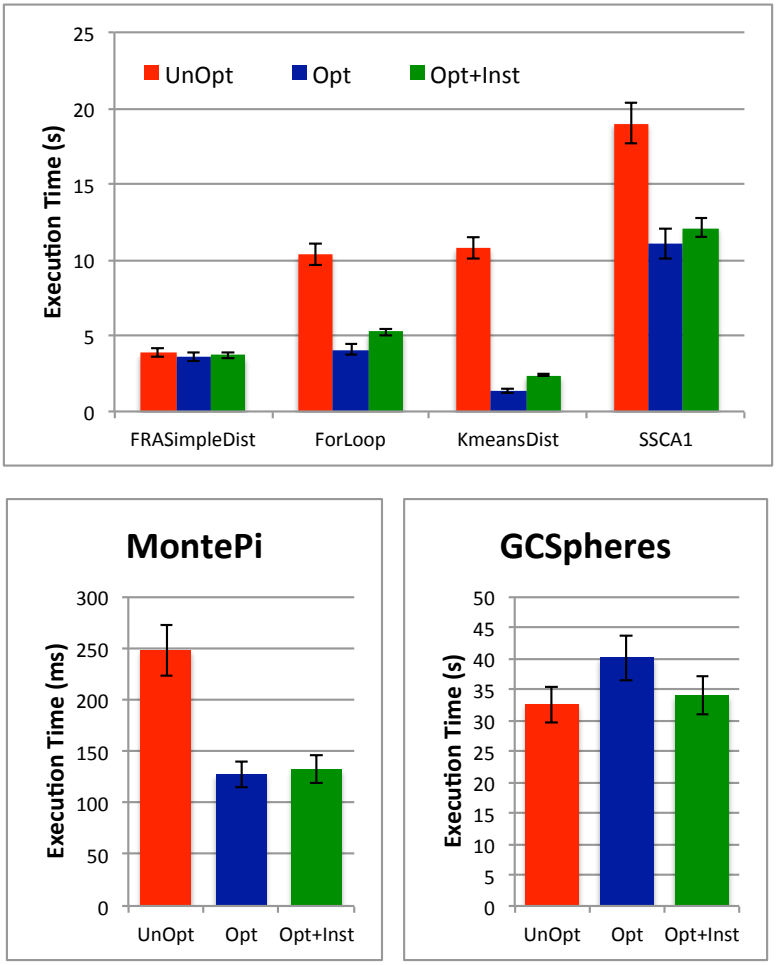


Figure 7: Execution time of the benchmarks for optimized and unoptimized versions. Shorter bars are better.

π. The optimized implementation uses `GlobalRef` to a shared variable at the root place. This variable holds the sum of local results from all places, instead of storing them in a distributed `PlaceLocalHandle` data structure. The optimized version also removed one instance of false place-shift. The benefit from the single place-shift optimization is not statistically significant, hence, it is not shown. All of the performance gain, *i.e.*, 48% over the unoptimized version, in MontePi stems from the use of `GlobalRef` to update a single variable rather than using `PlaceLocalHandle`.

GCSpheres was optimized to use a static nested `struct` instead of a static nested `class`. The optimized version is the StructSpheres benchmark. As evident from Figure 7, StructSpheres does not perform better than GCSpheres. GCSpheres creates several objects despite its use of `struct`. The garbage collector likely leads to high perturbation in execution time and thus, masks any performance improvement obtained using `struct`.

In summary, this experimental evaluation indicates that the performance gain due to each

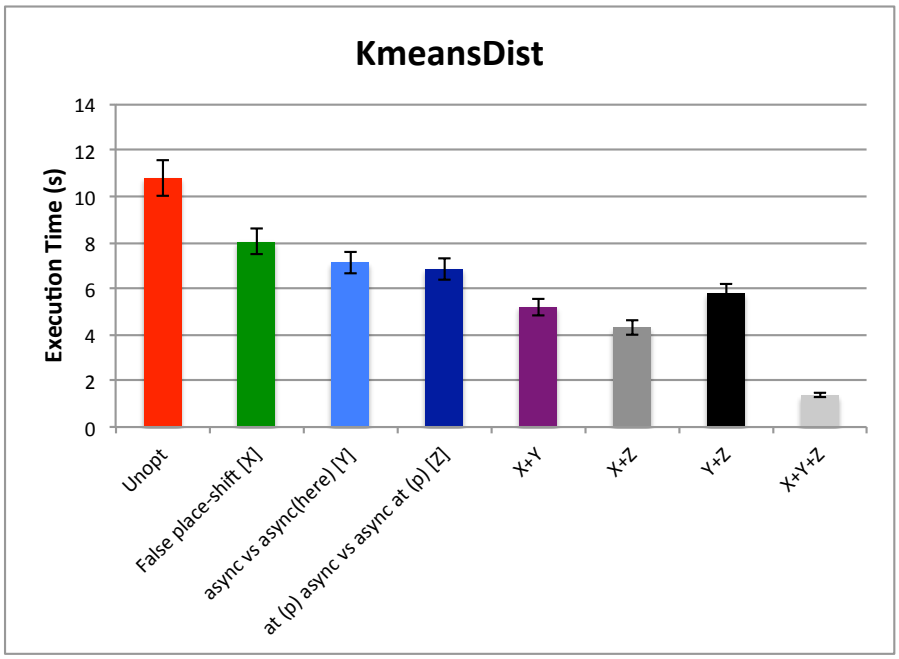


Figure 8: Breakdown of performance benefit from different optimizations on KMeansDist. Shorter bars are better.

of the code changes implemented after the identification of the opportunities by XAnalyzer depends not only on the benefit of each dynamic execution of the change, but also on the relative frequency of that section of code during execution. Further, the overall performance gains obtained by combining several optimizations is statistically significant and higher than those contributed by individual optimizations.

7.5 Performance Overhead due to Aspects

The third bars in the execution-time graphs in Figure 7, labelled as Opt+Inst, show the overhead of weaving aspects into user code. The perturbation due to aspect instrumentation is within 95% confidence interval of the execution time.

8 Conclusions and Future Work

XAnalyzer identifies and reports interesting patterns in X10 programs with the premise that offering high-level insights about a program's static source-code structure and its dynamic behaviour allows programmers to improve performance through code restructuring. Indeed, a number of program patterns pertaining to serialization, communication, spawning parallel tasks, and referencing shared variables are correctly identifiable by XAnalyzer in the set of benchmarks studied. Using these insights to performance tune the benchmarks removed several sources of overhead resulting in significant performance improvements.

Through pointcuts-and-advice-based instrumentation of programs, XAnalyzer provides a modular and an easy-to-understand implementation that has several opportunities for future

extension. To the best of our knowledge, XAnalyzer is the first framework that characterizes X10 programs in this way, and it offers insights for performance tuning.

Currently, XAnalyzer does not identify the `ateach` construct that provides efficient parallel iteration over distributed arrays. Future development will include support for characterization of additional X10 constructs that are critical to performance. A proper integration of XAnalyzer and existing development tools, such as X10DT, will make developing and performance tuning of X10 applications even more productive and fun.

References

[1] Dan Bonachea. GASNet Specification, v1.1. Technical report, Berkeley, CA, USA, 2002.

[2] Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne. PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010.

[3] J. Diamond, M. Burtscher, J.D. McCalpin, Byoung-Do Kim, S.W. Keckler, and J.C. Browne. Evaluation and Optimization of Multicore Performance Bottlenecks in Supercomputing Applications. In *International Symposium on Performance Analysis of Systems and Software*, pages 32–43, 2011.

[4] David Grove, Olivier Tardieu, David Cunningham, Ben Herta, Igor Peshansky, and Vijay Saraswat. A Performance Model for X10 Applications: What's Going On Under The Hood? In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 1:1–1:8, 2011.

[5] Gregor Kiczales and Erik Hilsdale. Aspect-Oriented Programming. In *8th European Software Engineering Conference*, ESEC/FSE-9, Vienna, Austria, 2001.

[6] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The Asynchronous Partitioned Global Address Space Model. In *Proceedings of the First Workshop on Advances in Message Passing*, AMP '10, 2010.

[7] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 Language Specification. http://x10.codehaus.org/x10/documentation.

[8] Hung-Hsun Su, Max Billingsley, and Alan D. George. Parallel Performance Wizard: A Performance System for the Analysis of Partitioned Global Address Space Applications. *International Journal of High Performance Computing Applications*, 24(4):485–510, November 2010.

[9] Hung-Hsun Su, Dan Bonachea, Adam Leko, Hans Sherburne, Max Billingsley, III., and Alan D. George. GASP! A Standardized Performance Analysis Tool Interface For Global Address Space Programming Models. In *8th International conference on Applied parallel computing: state of the art in scientific computing*, PARA'06, pages 450–459, Sweden, 2007.

[10] Nalini Vasudevan, Olivier Tardieu, Julian Dolby, and Stephen A. Edwards. Compile-time analysis and specialization of clocks in concurrent programs. In Oege Moor and MichaelI. Schwartzbach, editors, *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*, pages 48–62. 2009.

[11] X10-Team. X10 Development Tool. http://x10.sourceforge.net/x10dt/2.3.1/updateSite.

Programming Large Dynamic Data Structures on a DSM Cluster of Multicores*

Sai Charan Koduru, Min Feng and Rajiv Gupta

University of California, Riverside
email: {scharan, mfeng, gupta}@cs.ucr.edu

Abstract

Applications in increasingly important domains such as data mining and graph analysis operate on very large, dynamically constructed graphs, i.e. they are composed of dynamically allocated objects linked together via pointers. Parallel algorithms on large graphs can greatly benefit from software Distributed Shared Memory's (DSM) convenience of shared-memory programming and computational scalability of multiple machines. However, prior DSM systems did not provide programming and memory abstractions suitable for working with large dynamic data structures.

In this paper, we present *dynamic DSM* (dyDSM), an object-based DSM that targets applications which operate on large dynamic graphs. To achieve this goal, dyDSM differs from traditional DSM systems in four important ways. First, dyDSM provides programming abstractions that expose the structure of the distributed dynamic data structures to the runtime enabling the communication layer to efficiently perform *object level data transfers* including object prefetching. Second, the dyDSM runtime enables the management of large dynamic data structures by providing various memory allocators to support parallel initialization and storing of dynamic data structures like graphs. Third, dyDSM provides support for exploiting speculative parallelism that is abundant in applications with irregular data accesses. Fourth, dyDSM runtime actively exploits the multiple cores on modern machines to tolerate DSM access latencies for prefetching and updates. Our evaluation on a 40-core system with data mining and graph algorithms shows that dyDSM is easy to program in the very accessible C++ language and that its performance scales with data sizes and degree of parallelism.

1 Introduction

Clusters offer horizontal scalability both in terms of number of processing cores and the amount of memory. They are especially attractive for modern applications because, with the advances in network technology, a cluster can provide an application with a faster storage system than a disk based storage system [7]. While distributed memory clusters are more powerful than the shared memory machines they are composed of, programming them is challenging. While PGAS languages [14, 6, 12] ease distributed programming, they primarily explore parallelism for array-based data-parallel programs using data partitioning. For the most part they do not consider dynamic data structures nor do they explicitly support software speculation.

We have developed dyDSM which targets applications that employ large dynamic data structures and overcomes the inadequacies of prior DSM systems. Specifically, we target applications that operate on large dynamic data like graphs that exhibit no spatial locality and can exploit speculation for speedup. dyDSM addresses these issues by providing:

*This research was supported in part by a Google Research Award and NSF grants CCF-1318103, CCF-0963996 and CCF-0905509 to the University of California, Riverside.

1. Programming abstractions to communicate structure of the distributed data structure to the compiler;

2. Multiple allocators to perform parallel and local initialization and distribution of data structures;

3. Programming abstractions to effectively express speculation in programs and a runtime to transparently perform speculation on clusters; and

4. A runtime to actively exploit the multiple cores present on individual machines in the cluster.

Like PGAS languages, dyDSM automatically manages the distributed address space, but adds language and runtime support for speculative execution. We now describe the application features of interest and indicate the areas where prior solutions fall short.

Absence of Spatial Locality. Previous DSM systems (virtual memory and cache coherence protocol inspired systems [21] and Single-Program, Multiple Data [5] systems) were not designed to support dynamic data structures, did not support speculative parallelism, and did not consider multicore machines. They mostly targeted applications that use large arrays that exhibit spatial locality while the applications we study extensively use dynamically allocated data structures that lack spatial locality. Therefore caching and software cache coherence protocols [21] that they employ are ineffective in tolerating communication latency for the irregular applications we target.

dyDSM effectively deals with lack of spatial data locality in irregular applications: programming annotations expose the structure of the distributed dynamic data structures to the runtime enabling the communication layer to perform *object level data transfers* including object prefetching. By avoiding the use of traditional caching protocols, the need for cache coherence protocols and thus their drawbacks are eliminated.

Large Dynamic Data Structures. Previous object-based systems like Orca [1] required the programmer to explicitly and statically partition and allocate data across cluster. The computations were then executed on the site that contained the data. Given the absence of spatial locality in graph-like dynamic data structures, there is no intelligent way for the programmer to statically partition the data. Therefore, dyDSM takes this burden off the programmer by providing allocators that automatically and uniformly distribute the data across the cluster. Further, with *large* dynamic data structures, initializing the data in parallel is also very important. Otherwise, the data initialization could itself become a bottleneck. For this, dyDSM provides specialized allocators that enable parallel initialization and I/O of large data. Finally, dyDSM also provides an allocator to allocate data locally, in cases where the programmer knows in advance that some data is mostly used locally, and only occasionally needed on other machines in the cluster. The object transfers between thread-private memory and virtual shared memory are managed automatically by dyDSM.

Speculative Parallelism for Irregular Applications. Exploiting parallelism in irregular applications often also requires the use of speculation [26]. With irregular programs being our targeted applications, dyDSM eliminates the need for general purpose caching and coherence by employing *thread-private memory for the copy-in copy-out speculative model* [13]. The DSM is organized as a *two-level* memory hierarchy designed to implement the *copy-in copy-out* model of computation [13] that supports speculation but does not require use of traditional memory coherence protocols. As shown in Figure 1, the second level of DSM, named *virtual shared*

memory, aggregates the physical memories from different machines to provide a shared memory abstraction to the programmer. The first level of DSM consists of *virtual thread-private memories (PM)* created for each of the threads.

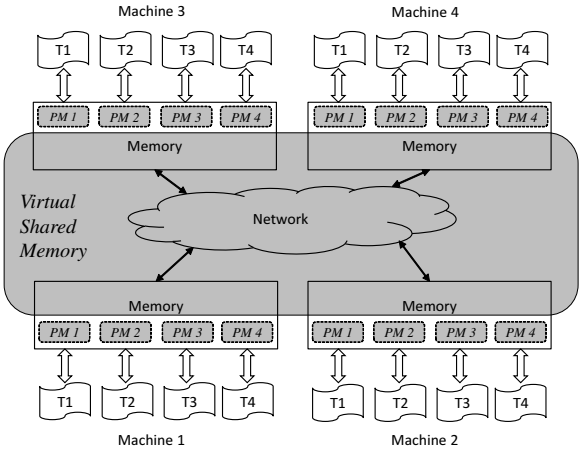


Figure 1: Memory hierarchy of dyDSM on a modern cluster.

To perform a computation, each thread must first *copy-in* the required data from the virtual shared memory into its virtual private memory. Then the thread can continue to use the local data copies in its private memory to perform computations. The *copy-out* of modified data from the private memory to the virtual shared memory makes it visible to all other machines. This computation model has two advantages. First, when a thread performs its computation using its private memory, it is able to exploit the temporal data locality *without generating network traffic*. Second, this model makes it easy to write programs that exploit *speculative parallelism* found in applications that employ dynamic data structures. Finally, the *copy-in copy-out* operations are introduced and performed by the dyDSM compiler and runtime, i.e. they do not burden the programmer. The programmer simply indicates the need for speculation by marking regions of code as *atomic* and dyDSM automatically handles all the above details.

Exploit Multicores for Tolerating Communication Latency. None of the previous DSM systems (eg. ORCA [1], Shasta [22] or Emerald [15]) were designed to utilize the modern multicore machines. Users can overcome this limitation by treating each processor core as a node in the DSM system and run multiple parallel computation tasks on each machine. However, due to the limited network resources on each machine, the performance may not scale well with the number of cores used from each machine. Since multicore machines are widely used today, dyDSM is designed to efficiently use of all of the available cores. Specifically, dyDSM utilizes a subset of available cores for asynchronous, off-the-critical-path communication, thus better hiding network latency than possible with previous approaches.

We have developed a prototype of the dyDSM system. The user writes programs in C/C++ augmented with dyDSM programming abstractions. The code is translated via dyDSM compiler based upon LLVM [19] to generate parallel code that calls the dyDSM runtime to effect allocation of objects in DSM and perform data transfers between thread-private memory and virtual shared memory. The runtime also implements speculation. We evaluated dyDSM on a cluster consisting of five eight-core machines using seven applications, including data mining applications (K-means Clustering, PageRank, Betweenness Centrality) and widely used graph algorithms (Delaunay Refinement, Graph Coloring, BlackScholes, Single Source Shortest Path).

These applications process large volumes of data. Our evaluation using 5 machine cluster with total of 40 cores yielded average speedup of 7x.

The rest of this paper is organized as follows. Section 2 describes our programming interface. Section 3 describes representation of dynamic data structures in DSM. Section 4 describes how dyDSM implements communications and supports speculation. Section 5 provides results of evaluation. Sections 6 and 7 discuss related work and conclusion.

2 Programming Interface of dyDSM

The programming interface of dyDSM does not place any significant burden on the programmer as it is like an interface for a shared memory system with some easy-to-use augmentations. dyDSM provides a familiar, shared memory-like API that allows creation and manipulation of dynamic data structures with ease. The augmentations perform the important task of identifying the presence and communicating the structure of a distributed dynamic data structure to the dyDSM runtime. They also indicate the form of parallelism, speculative or non-speculative. We now describe these programming extensions for use in C/C++ programs.

I. Structure declaration. Structure declarations are used to expose the details of the distributed dynamic data structures to the runtime enabling the communication layer to perform object level data transfers including object prefetching. In dyDSM, dynamic data structures are declared just like in shared-memory programs. The programmer need only add the `__dsm` annotation to the type declaration of an object type and the pointer variables that point to that object type. As an example, Figure 2 shows how a graph data structure must use annotation so that the created graph will reside in DSM. The `__dsm` annotation is used in two places: the definition of `struct Node`; and the array of pointers `neighbors[ ]` which links one node to its neighboring nodes in the distributed data structure. The same can be achieved if `Node` were defined as a `class`. While the annotation is simple to use for the programmer, the complex task of distributing the dynamic data structure is automatically carried out by the dyDSM runtime.

```
__dsm struct Node {
  int value;
  __dsm struct Node *neighbors[MAX_DEGREE];
};
```

Figure 2: Annotations for a distributed graph.

II. Dynamic allocation and deallocation. dyDSM eliminates the programmer burden of static partitioning and distribution of data across the cluster by providing allocators that automatically and uniformly distribute the data across the cluster. Further, with *large* dynamic data structures, dyDSM provides specialized allocators that enable parallel initialization and I/O of large data. Finally, dyDSM also provides an allocator to allocate data locally, in cases where the programmer knows in advance that some data is mostly used locally, and only occasionally needed on other machines in the cluster.

As in shared memory systems, in dyDSM, all annotated data structures are dynamically allocated and deallocated. Data distribution is achieved automatically at the allocation site by the dyDSM runtime. When an object is allocated at runtime, it is assigned to one of the machines in the cluster.

Table 1 shows the programming constructs provided by dyDSM for dynamically allocating and deallocating objects in DSM. They are similar to the `new` and `delete` operators in C++

Construct	Description	ID Format
`dsm_new Node;`	allocate an element of type `Node` and assign it to one of the machines	`[Node Type]-[Alloc. Type]-[Machine ID]-[Serial]`
`dsm_lnew Node;`	allocate an element of type `Node` and assign it to the local machine	`[Node Type]-[Alloc. Type]-[Machine ID]-[Serial]`
`dsm_cnew Node, id;`	allocate an element of type `Node` with `id` as its ID and assign it to one of the machines	`[Node Type]-[Alloc. Type]-[Customized ID]`
`dsm_delete pnode;`	deallocate the element referred to by pointer `pnode`	`--`

Table 1: Programming constructs for element allocation and deallocation.

except they allocate space in the DSM.The `dsm_new` construct randomly assigns the allocated element to one of the machines for load balancing. The `dsm_lnew` construct assigns the allocated element to the local machine. It is used to avoid unnecessary communication when the element is mostly accessed by the local machine. This is true if data is well partitioned among the machines. Construct `dsm_cnew` is designed to allow users to specify a customized id for an object. The last construct, `dsm_delete`, is used to deallocate the object referred to by the given pointer.

Next we illustrate the dynamic creation of the graph data structure declared in Figure 2 using `dsm_new`. We will illustrate the use of `dsm_lnew` and `dsm_cnew` in the next section. It is assumed that the graph is being read from a file and created and stored into the DSM. In Figure 3 the first loop (lines 4–8) reads the node information and allocates the nodes in dyDSM. The second loop (lines 9–12) reads the edge information and connects the nodes accordingly. Each invocation of `dsm_new` (line 6) assigns a given node to a random machine in the cluster. Array `node_ptrs` is a local variable that stores all node IDs, which are later used for adding edges.

```
01 // single thread builds the graph
02 if ( threadid == 0 ) {
03   begin_dsm_task(NON_ATOMIC);
04   for(i=0; i<nodes; i++) {    // read nodes
05     read_node(file, &id, &v);
06     node_ptrs[id] = dsm_new Node;
07     node_ptrs[id]→value = v;
08   }
09   for(i=0; i<edges; i++) {    // read edges
10     read_edge(file, &id1, &id2);
11     node_ptrs[id1]→nbrs.add(node_ptrs[id2]);
12   }
13   end_dsm_task();
14 }
```

Figure 3: Reading graph from a file and storing it in DSM.

III. Expressing Parallelism. Parallelism is exploited by creating multiple threads which execute the same code; however, they operate on different parts of the data structure. While this is similar to prior approaches for expressing parallelism, the difference in the code arises due to use of *copy-in copy-out* computation model [13]. Each thread executes blocks of code marked using `begin_dsm_task()` and `end_dsm_task()` in *copy-in copy-out* fashion. During the course of executing a code block the thread-private memory is used and any data not found

in the memory is copied into it. Thus the execution of a code block by a thread is carried out in *isolation* from other threads. At the end of the code block's execution the results are committed to the owners in virtual shared memory and made visible to all other threads. Note that programmer does not explicitly specify the data transfers but rather they are achieved via compiler introduced calls to the dyDSM runtime. Finally, the programmer can specify whether the update of shared memory must be `ATOMIC` or can be `NON_ATOMIC`. In the former case, atomic update guarantees sequential consistency at the code block level. It is used to implement speculative parallelism such that in case of misspeculation, commit fails and the execution of code block is reexecuted. The dyDSM runtime that implements virtual shared memory is responsible for checking atomicity.

```
01 // each thread works on a subset of nodes
02 for(i=tid*stride; i<(tid+1)*stride; i++) {
03   begin_dsm_task(ATOMIC);
04   nodeset = NULL;//construct subset of nodes
05   nodeset.insert(node_ptrs[i]);
06   while( (node = nodeset.next()) != NULL)
07     for(j=0; j<node→neighbors.size(); j++)
08       if ( check_cond(node→neighbors[j]) )
09         nodeset.insert(node→neighbors[j]);
10   update(nodeset); // update info.  in nodes
11   end_dsm_task();
12 }
```

Figure 4: Parallel computation using speculation.

Let us next illustrate the use of code blocks by threads and the use of `ATOMIC` and `NON_ATOMIC`. If we examine the example in Figure 3, because only a single thread constructs the graph, `NON_ATOMIC` is used. Figure 4 illustrates the use of `ATOMIC`. It shows a loop that performs computation on a pointer-based data structure. Similar loops can be found in many applications such as Delaunay refinement, agglomerative clustering [18], and decision tree learner [24]. In this loop, each thread executes a set of iterations of the outer for loop based upon its `tid`. A single iteration forms a code block that is executed atomically. During an iteration a node set is constructed from a given node in the data structure and then the computation performed updates the information in the node set. Although each iteration mostly works on different parts of the data structure, possible overlap between set of nodes updated by different threads (i.e., speculative parallelism) requires the use of `ATOMIC`. Note that it is difficult to write the above parallel loop using distributed memory programming model such as Message Passing Interface (MPI) since it requires developers to manually synchronize the pointer-based data structure across different memory spaces. On the other hand, dyDSM makes it easy to write the above parallel loop since the pointer-based data structure can be shared through reading and writing in the DSM.

3 Distributed Data Structures in dyDSM

In dyDSM, dynamic data structures are distributed in the virtual shared memory across multiple machines and objects on one machine can be logically linked to those on other machines. Moreover, as threads access and operate on objects of the data structure, these objects are copied into the thread-private memory. The same object may be present in the private memory of multiple threads. Figure 5 shows a graph data structure in virtual shared memory and copies of subsets of nodes in thread private memory on two machines. Let us discuss how a distributed data structure is represented and how the operations on it are implemented. The implementation of the dyDSM communication layer will be discussed in section 4.

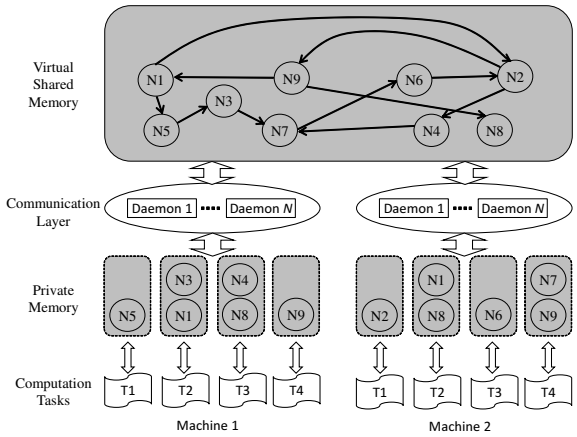


Figure 5: An example of dynamic data structure stored in dyDSM.

The compiler translates the annotated data structures and pointers by introducing calls to dyDSM runtime at appropriate places so that data needed by a thread is made available in its thread-private memory via data transfers performed by dyDSM runtime. The compiler instruments each annotated data structure to ensure when an object is accessed, the runtime is called to check if the object is in the thread-private memory. If not, data transfer is performed to copy the data object from the remote memory to the thread-private memory. All subsequent accesses to the same object will be directed to the copy in the private memory.

The dynamically allocated objects can only be accessed via pointers. Each object is assigned a unique ID that acts as its virtual address and is used to access it. Thus, the pointers are merely represented as IDs of objects they point to. When an object is allocated, its ID is generated by the runtime on the machine where the allocator is called. In dyDSM, an ID of an object is a string consisting of three parts: *object type*; *allocation type* used to allocate the object; and a *unique string* generated according to the allocation policy. The runtime fills the first two parts at the time of allocation. The unique string is determined according to the allocation mechanism used.

The construction of the unique string according to the allocation mechanism used is shown in Table 1 (third column). When `dsm_new` and `dsm_lnew` are used, the unique string is a combination of the machine ID where the allocator is called and a unique number which is generated using a local counter. When construct `dsm_cnew` is used, the unique string in the ID is specified by the programmer. This construct is designed for expressing dynamic arrays. Accessing the element using the customized ID is similar to accessing an array element in C/C++, where the structure type and customized ID are used as the array name and index.

When an ID is dereferenced, information in it is used to locate the object and then the object is copied into the thread-private memory. When committing data to shared memory, the runtime uses the object IDs to locate the owner machine. For different types of IDs (`n`, `l`, or `c`), different schemes are used to locate the machine where the object is stored. For object allocated using `dsm_new` (i.e., `n` type ID), the machine is located by hashing the ID. For object allocated using `dsm_lnew` (i.e., `l` type ID), the machine is located using the machine ID stored in the object ID. For an object allocated using `dsm_cnew` (i.e., `c` type ID), the machine is located by hashing the object ID.

In Figure 3, we demonstrated how a single thread can read a graph from a file build it using `dsm_new` and store it in DSM. Now, we demonstrate how the same task can be performed in

parallel using `dsm_cnew`. We assume that the input file is replicated on all machines. Each thread reads the entire file; however, it only builds part of the graph. In Figure 6, in the first loop (lines 3–7) each thread reads the node information and assigns each node a customized ID using `dsm_cnew`. In the second loop (lines 12–19), each thread updates the edges in the graph using the customized node IDs. With the use of customized IDs, we avoid using the array `node_ptrs` used in Figure 3 that may not fit in the local memory if the graph is extremely large. As we can see from line 12, the update of edges requires use of `ATOMIC` as multiple threads may be adding edges to the same node.

```
01 begin_dsm_task(NON_ATOMIC);
02 // read nodes
03 for(i=0; i<(threadid+1)*stride1; i++) {
04   read_node(file, &id, &v);
05   if ( i >= threadid*stride1 ) {
06     dsm_cnew Node, id;
07     Node[id].value = v;
08   }
09 }
10 end_dsm_task();
11 // read edges
12 for(i=0; i<(threadid+1)*stride2; i++) {
13   read_edge(file, &id1, &id2);
14   if ( i >= threadid*stride2 ) {
15     begin_dsm_task(ATOMIC);
16     Node[id1].nbrs.add(&Node[id2]);
17     end_dsm_task();
18   }
19 }
```

Figure 6: Reading and storing a graph in DSM in parallel via speculation.

While the code shown in Figure 6 requires use of `ATOMIC`, next we show how to perform parallel graph construction without use of `ATOMIC` using `dsm_lnew`. As before, in this version we assume that the input file is replicated on all machines. Each thread reads the file and constructs part of the graph identified to belong to its `partition`. In Figure 7 a dynamic array is created using `dsm_cnew` for holding pointers to nodes. It is used by all threads to locate nodes. However, `dsm_lnew` is used to allocate nodes. All nodes are allocated locally on the machine where they belong. When we update nodes by adding edges, each thread only updates nodes in its own partition. Thus, not only do we avoid the use of speculation, we also achieve greater efficiency due to locality.

4 Data Communication in dyDSM

Next we describe the communication layer of dyDSM and show how it implements *data transfers* and *performs communication in parallel with computation.* The communication layer also supports prefetching to further hide network latency and implements atomic updates of DSM to support speculation.

I. Communication exploiting multicore machines. In dyDSM, each multicore machine runs multiple computation threads. The performance sometimes does not scale well with the number of parallel computation tasks on each machine. To improve performance in this case, dyDSM uses a communication layer consisting of a multiple communication daemons that transfer data between the thread-private memory and the virtual shared memory (see Figure 5). The

```
01 // read nodes
02 begin_dsm_task(NON_ATOMIC);
03 for(i=0; i<nodes; i++) {
04   read_node(file, &id, &v);
05   if ( partition(id) == threadid ) {
06     dsm_cnew NodePtr, id;
07     NodePtr[id] = dsm_lnew Node;
08     NodePtr[id]→value = v;
09   }
10 }
11 end_dsm_task();
12 // read edges
13 begin_dsm_task(NON_ATOMIC);
14 for(i=0; i<edges; i++) {
15   read_edge(file, &id1, &id2);
16   if ( partition(id1) == threadid )
17   node_ptrs[id1]→nbrs.add(NodePtr[id2]);
18 }
19 end_dsm_task();
```

Figure 7: Reading a graph and storing it in DSM in parallel without using speculation.

daemons run on underutilized cores, helping the computation threads perform communication. The communication is moved off the critical path of a computation thread by offloading it to daemon threads. Thus, computation and communication is performed in parallel for improved performance.

The communication layer works as follows. On a thread's first reference to an object in virtual shared memory, the communication layer copies the object from the virtual shared memory to the thread's private memory. Subsequent accesses to the object, including both read and write, refer to the object copy in the private memory. Thus, the thread manipulates the object in *isolation*, free of contention with other threads. More importantly, caching the object eliminates the communication overhead of subsequent access to the same object. When a thread calls `dsm_end_task()`, the communication layer commits all modified objects in the private memory to the virtual shared memory allowing other threads to see the updates.

The communication daemons perform data communication in parallel with computation. When a thread needs an object, one of the daemons fetches the object from the shared memory for it. After the thread gets the object and continues its computation, the daemon can continue to prefetch the objects that will be probably accessed in the future. This can ensure that the data required by the thread is *mostly* available locally when needed. When a thread calls `dsm_end_task()`, one of the daemon also helps it perform commit so that the computation thread can continue to the next computation. Thus, the communication layer hides the network latency from the threads. As a result, the threads only need to operate on data in thread-private memory. Communication layer hides *read latency* by prefetching data needed by threads and hides *write latency* by performing commit in parallel with computation.

II. Adapting the number of communication daemons. The number of communication daemons directly impacts the performance of the system. Figure 8 shows the performance of two benchmarks with different number of communication daemons. The execution time is normalized to the one with one communication daemon. More communication daemons in graph coloring help move more communication off the critical path. However, increasing the number of communication daemons also leads to consuming more computing resources and eventually slows down the performance. For SSSP, one daemon is the best choice. We use a dynamic scheme to find the minimal number of communication daemons to use.

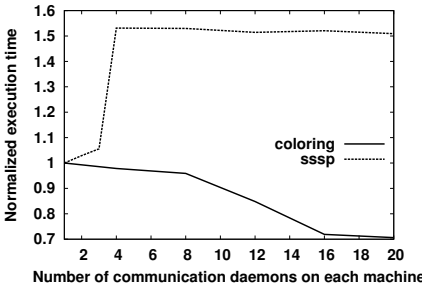


Figure 8: Performance with varying number of communication daemons.

In dyDSM, a queue between the computation threads and the communication layer holds all pending communication requests from computation threads. The queue length is used as a hint to pick the number of daemons. Large queue length means that outstanding communication requests get queued, degrading performance. Longer queue can also cause higher misspeculation rates due to delay in misspeculation detection. On the other hand, empty queue could indicate that the number of communication daemons is more than needed, wasting computing resources. In dyDSM, the program always starts with a small preset number of communication daemons. Each time a communication daemon processes a communication request, it checks the queue length. If the queue length is larger than the number of daemons, dyDSM increases the number of communication daemons by a predefined stride. If the queue length is smaller than the number of communication daemons, the number of communication daemons is decreased. To reduce the overhead of starting and terminating daemons, our implementation puts extra daemons to sleep rather than terminating them.

III. Data prefetching. When a computation task wants to access an object, it first checks whether a copy already resides in the thread-private memory. If not, the computation task waits for the remote access to fetch the data object. To avoid this wait, the communication daemons prefetch data for the computation tasks. To perform data prefetching, when a data element is accessed by a computation task for the first time, a communication daemon also predicts what elements are likely to be accessed next. If the predicted elements have not been previously copied into the local virtual memory, then the daemon sends data requests to their owners. The prediction is implemented similar to an event handler. To enable data prefetching, programmer must register a *prediction function* for the dynamic data structure. The dyDSM construct used to register the prediction function identifies the data structure requiring prefetching and the *recursion limit* (or *depth*) up to which the prediction is to be performed. When an element of the data structure is accessed, a communication daemon calls the prediction function, which takes the accessed element as input and fetches the elements that are likely to be accessed.

```
__dsm_predict(struct Node, 2)
void prediction_node(struct Node *pnode) {
   for (int i=0; i<MAX_DEGREE; i++)
     dsm_fetch(pnode→nbrs[i]);
}
```

Figure 9: An example of user-defined prediction function.

Figure 9 shows an example of a user-defined prediction function for the graph data structure used in the example from preceding section. The prediction function prefetches all 2-hop neighbors of the accessed node. This can be used in computations working on connected subgraphs,

which is true in many graph applications (e.g., Delaunay Refinement, Betweenness Centrality, Coloring).

IV. Speculation in dyDSM. Many parallel programs require speculation to perform computation correctly. For example, in graph coloring, each computation calculates the color of one node using the colors of its neighbors. It is possible for two parallel threads to read and write the color of one node at the same time; thus causing two neighboring nodes to have the same color. With speculation, all threads are assumed to access different nodes. They perform computation in isolation. When two threads actually access the same node, one thread succeeds in committing its results while the other fails and is reexecuted. With speculative execution, we also achieve sequential consistency at coarse-grained level of code regions.

Speculation requires threads to perform computation atomically and in isolation. In dyDSM, isolation comes for free since our communication layer and the use of thread-private memory already ensure computations are performed in isolation. Therefore, speculative execution does not affect the computation between `begin_dsm_task()` and `end_dsm_task()`. To ensure atomicity, speculation requires the commit to be performed atomically. When speculation is enabled, the communication layer performs commit as follows: (i) First, the daemon sends request to the virtual shared memory to lock all objects marked as read in the private memory. If locking fails for any object, it releases all acquired locks and then retries locking again. (ii) After successful locking, the daemon compares the versions of all read objects in the private memory with their versions in the virtual shared memory. If the versions do not match, it means that one of the read objects must have since been updated by another thread in the virtual shared memory. Therefore, the computation must be reexecuted. (iii) If versions match, the daemon then commits all written objects to the virtual shared memory and increases their version numbers by one. (iv) Finally, the daemon sends request to the virtual shared memory to unlock all acquired locks so that these objects can be updated by other threads. With remote communication, the commit for speculation can be time-consuming. However, dyDSM overcomes this problem by making the communication layer perform the commit in parallel with the computation; thus, hiding its overhead from computation.

5 Evaluation

This section evaluates dyDSM. The experiments were conducted on a cluster consisting of five eight-core (2×4-core AMD Opteron 2.0GHz) DELL PowerEdge T605 machines. These machines are running the *Ubuntu 10.04* operating system. They are connected through a Cisco ESW 540 switch, which is a 8-port 10/100/1000 Ethernet switch. Throughout the experiments, dyDSM always uses the memory of all five machines no matter how many machines are used for computation.

Implementation of dyDSM. Figure 10 shows an overview of our prototype implementation of dyDSM. It consists of a source-to-source compiler and a runtime library. The source-to-source compiler is implemented using the LLVM compiler infrastructure [19]. It instruments the annotated distributed data structures, translates pointers to these data structures, and generates code for prefetching. The dyDSM runtime is implemented on top of *Memcached*, an open-source distributed memory object caching system. The runtime provides support for communication, memory allocation, and speculation. In the experiments, the communication layer on each machine consists of 32 daemons. By creating many daemons dyDSM provides high network throughput. We cannot create greater number of daemons on one machine due to the size of Memcached connection pool. When no data needs to be transferred, the daemons are blocked and thus consume no processing resource.

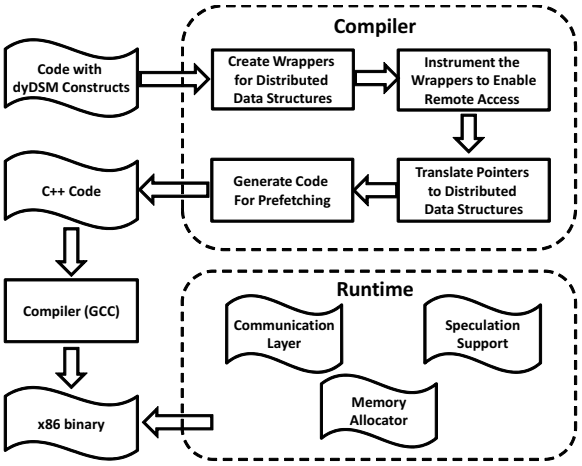


Figure 10: Overview of dyDSM prototype.

Benchmarks. To evaluate our approach, we performed experiments using seven applications, five of which require speculation. Since dyDSM is built for applications with large data sets, we selected these applications as they are data-intensive. Some of them are widely-used data mining programs designed to process large volume of data while others are graph algorithms that are used in many domains. rtabtab:bench lists the applications used, the key data structure involved and whether a speculative implementation is used. Next, we briefly describe these applications.

Benchmark	Data Struct.	Speculation?
Delaunay Refinement	Graph	Yes
Graph Coloring	Graph	Yes
Betweenness Centrality	Graph	Yes
K-means Clustering	Dynamic Array	Yes
PageRank	Graph	No
BlackScholes	Dynamic Array	No
Single Source Shortest Path (SSSP)	Graph	Yes

Table 2: Benchmark summary.

`Delaunay Refinement` implements mesh generation that transforms a planar straightline graph to a Delaunay triangulation of only quality triangles. The data set is stored in a pointer-based graph structure. `Graph Coloring` is an implementation of the scalable parallel graph coloring algorithm proposed in [4]. Each computation task colors one node using the information of its neighbors. `K-means Clustering` [10] is a parallel implementation of a popular clustering algorithm that partitions n points into k clusters. `PageRank` is a link analysis algorithm used by Google to iteratively compute weight for every node in a linked webgraph. `BlackScholes` taken from the PARSEC suite [2] uses partial differential equation to calculate the prices of European-style options. `Betweenness Centrality` is a popular graph algorithm used in many areas. It computes the shortest paths between all pairs of nodes. `SSSP` implements a parallel algorithm for computing shortest path from a source node to all nodes in a directed graph. The implementation is based on the Bellman-Ford algorithm.

5.1 dyDSM Scalability with Parallelism

Applications with high degree of parallelism can take advantage of additional cores available on multiple machines. However, the additional threads created to exploit more parallelism also stress the dyDSM communication layer. Next we study how the application speedup scales as greater number of threads are created to utilize more cores. The threads created can be scheduled on available cores in two ways: (distributed) they can be distributed across all machines; or (grouped) they can be grouped so that they use minimal number of machines. Figure 11 shows the application speedups for both the scenarios while the accompanying Table 3 summarizes the fastest execution times. The baseline is the sequential version of the application. Comparing the two curves, we observe that distributing the computation threads across the machines is a better scheduling strategy. For all benchmarks except `blackscholes`, the speedup achieved by distributing threads rises faster as the number of computation threads is increased. By distributing computation threads across the machines, idle cores left on each machines can be used by the communication layer. Therefore, the computation threads do not compete with the communication layer for cycles. For `blackscholes`, grouping threads achieves better performance because of high degree of data sharing among threads.

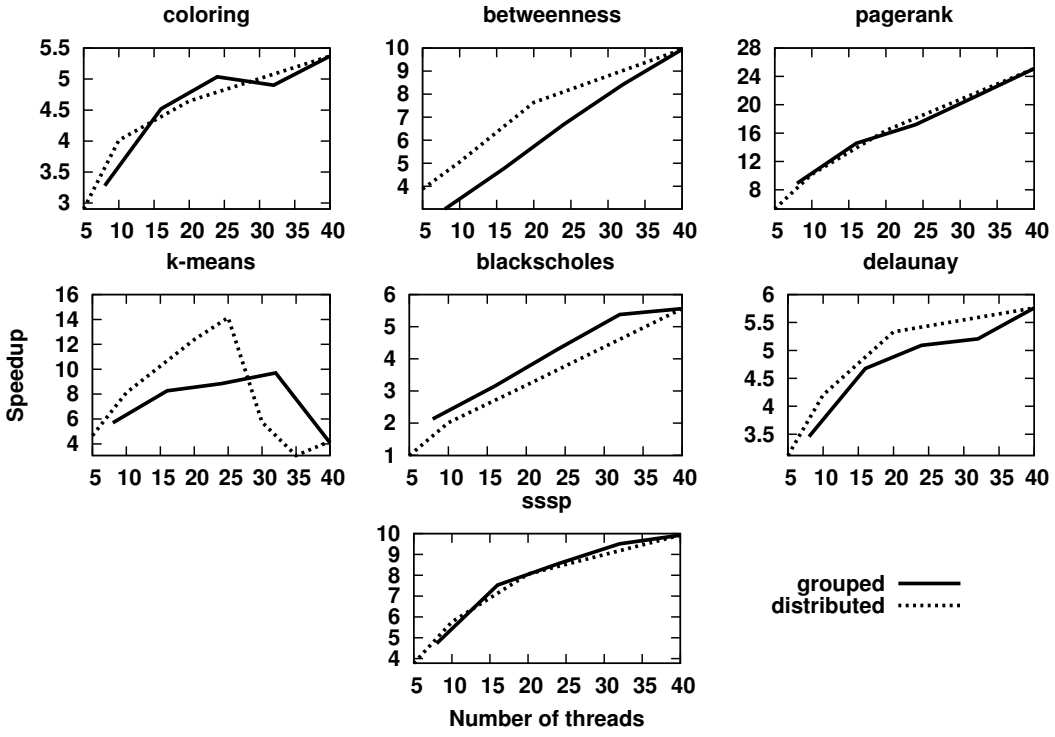


Figure 11: Speedups with varying number of computation threads.

We observe that the performance of most applications scales well with the number of computation threads. The parallel versions are always faster than the sequential version with maximum speedup of 25x. The performance of `k-means` drops with more than thirty computation threads due to very high lock contention in atomicity checks. On average, we achieve 7x speedup for the benchmarks using forty computation threads on five machines, which demon-

Benchmark	Execution time [fastest] (seconds)	Sequential time (seconds)
Delaunay Refinement	1886.097	39532.22
Graph Coloring	1566.785	23214.378
Betweenness Centrality	25432.078	378835.65
K-means Clustering	16562.046	176791.708
PageRank	1678.381	48018.028
BlackScholes	25789.837	182441.231
Single Source Shortest Path	15067.332	197785.465

Table 3: Execution times at maximum speedup for the distributed execution configuration.

strates dyDSM's ability to handle parallel applications running across multiple machines. For six out of seven benchmarks, we achieve highest speedups when eight computation threads are run on each machine. This shows that the dyDSM communication layer utilizes the cores only to a limited extent; thus leaving them free to be used by the computation threads.

5.2 Performance of the Adaptive Scheme

Benchmark	Adaptive / Static
Delaunay Refinement	1.10
Graph Coloring	0.66
Betweenness Centrality	1.04
K-means Clustering	1.04
PageRank	1.22
BlackScholes	1.19
Single Source Shortest Path	1.54

Table 4: Execution time: Adaptive vs. Ideal static scheme.

This section evaluates the performance of the dynamic scheme that adapts the number of communication daemons at runtime (described in Section 4). Table 4 shows the execution time of the adaptive scheme normalized to the ones of the ideal static scheme. For the adaptive scheme, we set the initial daemon number to four. For ideal static scheme, we found the number of daemons that gave the best performance. From the table, we can see that the adaptive scheme greatly improves the performance over the best static scheme for benchmark `graph coloring`; this is because the number of chosen daemons varies over the program lifetime and this is aided by the dynamic scheme. We observe that the performance of `SSSP` is most degraded: in the static scheme `SSSP` performs best at 1 communication thread. The dynamic scheme performs worse because it favors more communication threads and therefore rarely runs at the 1-communication thread configuration. For the remaining benchmarks, the adaptive scheme performs close to that of the static scheme. The slight degradation can be attributed to the overhead of adaptation: periodically measuring the length of queue, waking up more communication threads, etc.

6 Related Work

Many previous works have been done on DSM systems [5, 22, 23]. However, *these DSM systems were not developed for modern clusters, where each node has many processing cores.* They do not make use of the multiple cores on each node. In addition, these DSM systems work poorly

for applications that use using dynamic features provided by modern programming languages. The performance of SPMD-based DSM is greatly degraded for modern applications as these dynamic features make the appropriate distribution of data impossible to determine at compile-time. In a cache-coherent DSM system, data needs to be replicated and/or migrated on-the-fly as it is impossible to statically figure out the memory access patterns for programs using dynamic features. dyDSM overcomes these drawbacks. We allow automatic data distribution, while keeping the communication overhead off the critical path by exploiting the multiple cores on each machine. Recently, software transactional memory has been introduced as a parallel programming paradigm for clusters [20, 3, 16, 17, 9]. Compared to them, we move a step further in hiding latency. Our communication mechanism makes efficient use of multicore machines. Latency tolerance is achieved by moving communication off the critical path, i.e., commits occur asynchronously in parallel with computations.

Partitioned global address space (PGAS) is a parallel programming model which tries to combine the performance advantage of MPI with the programmability of a shared-memory model. It creates a virtual shared memory space and maps a portion of it to each processor. PGAS provides support for exploiting data locality by affining each portion of the shared memory space to a particular thread. The PGAS programming model is used in UPC [8], Co-Array Fortran [11], Titanium [14], Chapel [6] and X10 [12]. UPC, Co-Array Fortran, and Titanium use SPMD style with improved programmability. Chapel and X10 extend PGAS to allow each node to execute multiple tasks from a task pool and invoke work on other nodes. These PGAS programming models primarily explore parallelism for array-based data-parallel programs using data partitioning. For the most part they do not consider dynamic data structures.

Given this works' focus on speculative parallelism for applications with low, non-deterministic sharing, we do not expect much benefit from sophisticated load balancing schemes as proposed in [25]. Moreover, at today's scale, such schemes can easily overwhelm the network. For purposes of this work, Memcached's data partitioning along with on-demand prefetching suffice.

7 Conclusion

This paper presented dyDSM, which is a DSM system designed for modern dynamic applications and clusters composed of multicore machines. It provides a simple to use programming interface and a powerful runtime and compiler that handles of the tedious tasks of using distributed-memory. dyDSM's support for large dynamic data structures makes it unique and very relevant for modern applications. The communication layer of dyDSM runtime makes use of multiple cores on machines and prefetching to high network latency. dyDSM also offers sequential consistency at coarse-grain level and supports speculative parallelism. Our evaluation of dyDSM on a cluster of five eight-core machines shows that it performs well giving an average speedup of 7x across seven benchmark programs.

References

[1] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *TSE*, 18:190–205, 1992.

[2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, pages 72–81, 2008.

[3] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP*, pages 247–258, 2008.

[4] Erik B. Boman, Doruk Bozdağ, Unit Catalyurek, Assefaw H. Gebremedhin, and Fredrik Manne. A scalable parallel graph coloring algorithm for distributed memory computers. In *EURO-PAR*, pages 241–251, 2005.

[5] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *J. Supercomputing*, 2(2):151–169, 1988.

[6] Bradford Chamberlain, David Callahan, and Hans Zima. Parallel programmability and the chapel language. *IJHPCA*, 21(3):291–312, 2007.

[7] Jeffrey S. Chase, Darrell C. Anderson, Andrew J. Gallatin, Alvin R. Lebeck, and Kenneth G. Yocum. Network i/o with trapeze. In *HOTI*, 1999.

[8] UPC Consortium. UPC language specifications, v1.2. *Lawrence Berkeley National Lab Tech Report LBNL-59208*, 2005.

[9] Alokika Dash and Brian Demsky. Integrating caching and prefetching mechanisms in a distributed transactional memory. *TPDS*, 22(8):1284–1298, 2011.

[10] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Workshop on Large-Scale Parallel KDD Systems*, 1999.

[11] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A multi-platform co-array fortran compiler. In *PACT*, 2004.

[12] Philippe Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[13] Min Feng, Rajiv Gupta, and Yi Hu. SpiceC: scalable parallelism via implicit copying and explicit commit. In *PPoPP*, pages 69–80, 2011.

[14] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su, and K. Yelick. Titanium language reference manual. *U.C. Berkeley Tech Report, UCB/EECS-2005-15*, 2005.

[15] E Jul, H Levy, N Hutchinson, and A Black. An object-oriented language and system that indirectly supports dsm through object mobility. 1988.

[16] Christos Kotselidis, Mohammad Ansari, Kimberly Jarvis, Mikel Lujan, Chris Kirkham, and Ian Watson. DiSTM: A software transactional memory framework for clusters. In *ICPP*, pages 51–58, 2008.

[17] Christos Kotselidis, Mohammad Ansari, Kimberly Jarvis, Mikel Lujan, Chris Kirkham, and Ian Watson. Investigating software transactional memory on clusters. In *IPDPS*, pages 1–6, 2008.

[18] Milind Kulkarni, Martin Burtscher, Calin Casaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, 2009.

[19] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[20] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP*, pages 198–208, 2006.

[21] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Concurrency*, 4(2):63–79, 1996.

[22] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. In *ASPLOS*, pages 174–185, 1996.

[23] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *ASPLOS*, pages 297–306, 1994.

[24] Paul E. Utgoff, Neil C. Berkman, and Jeffery A. Clouse. Decision tree induction based on efficient tree restructuring. *Mach. Learn.*, 29(1):5–44, 1997.

[25] Robert P Weaver and Robert B Schnabel. Automatic mapping and load balancing of pointer-based dynamic data structures on distributed memory machines. In *Scalable High Performance Computing Conference, 1992. SHPCC-92. Proceedings.*, pages 252–259. IEEE, 1992.

[26] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for speculative parallelization of partially-parallel loops in dsm multiprocessors. In *HPCA*, pages 135–139, 1999.

An Efficient Implementation of Stencil Communication for the XcalableMP PGAS Parallel Programming Language

Hitoshi Murai[1] and Mitsuhisa Sato[2]

[1] RIKEN AICS, Kobe, Japan
h-murai@riken.jp
[2] Center for Computational Science, University of Tsukuba, Tsukuba, Japan
msato@cs.tsukuba.ac.jp

Abstract

Partitioned Global Address Space (PGAS) programming languages have emerged as a means by which to program parallel computers, which are becoming larger and more complicated. For such languages, regular stencil codes are still one of the most important goals. We implemented three methods of stencil communication in a compiler for a PGAS language XcalableMP, which are 1) based on derived-datatype messaging; 2) based on packing/unpacking, which is especially effective in multicore environments; and 3) experimental and based on one-sided communication on the K computer, where the RDMA function suitable for one-sided communication is available. We evaluated their performances on the K computer. As a result, we found that the first and second methods are effective under different conditions, and selecting either of these methods at runtime would be practical. We also found that the third method is promising but that the method of synchronization is a remaining problem for higher performance.

1 Introduction

As computer systems become larger and more complicated, for example, with respect to memory hierarchy and interconnect topology, to achieve higher performance, a programming method that can provide users with high productivity and high performance is strongly demanded. Partitioned Global Address Space (PGAS) programming languages, such as XcalableMP (XMP) [22], the coarray feature of Fortran 2008 [18], Unified Parallel C (UPC) [21], Chapel [8], and X10 [5], are considered to meet this demand and have been investigated extensively.

A number of PGAS languages set the goal of supporting a broader range of applications, such as irregular applications having task parallelism that High Performance Fortran (HPF) [14], which is an ancestor of PGAS languages, could never support successfully. However, the situation whereby regular stencil codes, such as reported in [19, 10, 7], are among the most significant goals remains unchanged. Therefore such languages should provide a means for effectively handling stencil communication.

We implemented three types of stencil communication in the Omni XcalableMP compiler that we are currently developing, and the details of these types of stencil communication are presented herein. The first type is based on the derived datatype of Message Passing Interface (MPI) [15]. The second type is based on packing/unpacking buffers, which may be executed in parallel if possible. Finally, the third type is experimental and is based on the extended RDMA interface [9] dedicated for the K computer [16]. The goal of the present study is to explore an optimal method of implementing stencil communications in compilers for PGAS languages.

The contributions of the present paper include:

- Three implementations of stencil communication, including an RDMA-based implementation, for PGAS language compilers, are described.
- Their advantages and disadvantages are discussed based on their evaluation on the K computer.

The remainder of the present paper is organized as follows. Sections 2 and 3 provide a brief overview of the XMP language specification and the Omni XMP compiler, respectively. Sections 4 and 5 describe the proposed implementations of stencil communication, which are evaluated in Section 6. After discussing related research in Section 7, Section 8 presents the conclusion and areas for the future research.

2 XcalableMP

XcalableMP (XMP) is a directive-based language extension for Fortran and C, proposed by the XcalableMP Specification Working Group. XMP supports typical parallelization methods based on the data/task parallel paradigm under the "global-view" model, and enables parallelization of the original sequential code with minimal modification. XMP also includes the coarray feature imported from Fortran 2008 for "local-view" programming. In addition, the combination of OpenMP directives and XMP is to be included in the next update of its specification. In this section, we present a brief overview of the specification of XMP.

The readers can find an example of an XMP program in Figure 8.

2.1 Execution and Memory Model

Execution Model The execution entities in an XMP program are referred to as *XMP nodes* or, more simply, *nodes*. An XMP node is mapped at runtime to a physical computation node on which an MPI process can run with multithreading in hybrid parallelization or with multiple MPI processes in flat parallelization.

The basic execution model of XMP is Single Program Multiple Data (SPMD). Each XMP node starts execution from the same main routine and continues to execute the same code independently (i.e., asynchronously) until an XMP directive, which is *global* and to be executed collectively by all of the nodes, is encountered.

Memory Model Each node has its own memory and can directly access only data contained therein. If a node should access data on a remote node, users must explicitly specify an inter-node communication with an XMP directive, such as `reflect` described in the following section, in global-view programming or coarrays in local-view.

2.2 Data and Work Mapping

Data Mapping First, an array is *aligned* with a *template*, which is a virtual array, by the `align` directive. Next, the template is *distributed* onto a *node set* in a certain format, such as the block format, the cyclic format, or the block-cyclic format, by the `distribute` directive. As a result, each element of the array is assigned through the distributed template to one or more nodes (Figure 1). The set of local elements of an array logically form a rectangle and is allocated in the local memory.

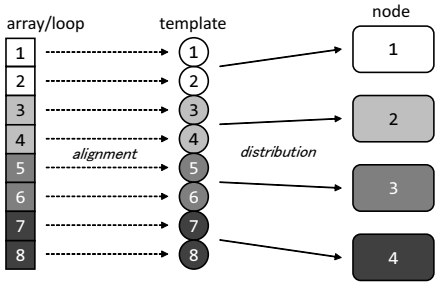


Figure 1: Data and Work Mapping in XMP

Work Mapping An iteration space of a loop nest is, in analogy with an array, "aligned" with a template by the `loop` directive. An aligned loop nest is executed in parallel by the executing nodes.

2.3 Directives for Stencil Communication

2.3.1 The `shadow` Directive

An array distributed in the block or non-uniform block ("gblock") format may have an additional area referred to as *shadow*, which is used as a buffer to communicate with the neighbor elements of each block of the array.

Figure 2 (a) shows the syntax of the `shadow` directive of XMP, which is used to specify the width of the shadow area of each axis of an array[1]. Users can also specify different widths for the lower and upper shadows of an axis.

2.3.2 The `reflect` Directive

Figure 2 (b) shows the syntax of the `reflect` directive of XMP, which is used to update the shadow area of an array with the value of its corresponding reflection source.

Specifying the `width` clause, only a part of the shadow area can be updated. In addition, when the `/periodic/` modifier is specified in the `width` clause, the update is "periodic" along the axis, which means that the shadow object at the global lower (upper) bound is updated with the value of the data object at the global upper (lower) bound.

A communication induced by the `reflect` directive can be *asynchronous* when the `async` clause is specified with the directive. Such asynchronous communications are issued but not completed, along with nonblocking communications of the MPI standard, at the point of the directive to overlap with the following computation.

Figure 3 illustrates how the `shadow` and `reflect` directives work for a one-dimensional array.

2.3.3 The `wait_async` Directive

The `wait_async` directive (Figure 2 (c)) blocks and therefore statements following it are not executed, until all of the asynchronous communications specified by *async-id*s are complete.

[1]When *shadow-width* is of the form "`*`", the entire area of the array is allocated on each node, and all of the area not owned by it is regarded as shadow. This feature is referred to as "full shadow" but is not dealt with in the present paper.

```
[F]   !$xmp shadow array-name ( shadow-width [, shadow-width]... )
[C]   #pragma xmp shadow array-name [shadow-width][[shadow-width]]...
```

where *shadow-width* must be one of:

int-expr
int-expr : *int-expr*
*

(a) the `shadow` directive

```
[F]   !$xmp reflect ( array-name [, array-name]... ) &
                                    & [width ( reflect-width [, reflect-width]... )] [async ( async-id )]
[C]   #pragma xmp reflect ( array-name [, array-name]... ) &
                                    & [width ( reflect-width [, reflect-width]... )] [async ( async-id )]
```

where *reflect-width* must be one of:

[/periodic/] int-expr
[/periodic/] int-expr : *int-expr*

(b) the `reflect` directive

```
[F]   !$xmp wait_async ( async-id [, async-id ]... ) [on nodes-ref | template-ref]
[C]   #pragma xmp wait_async ( async-id [, async-id ]... ) [on nodes-ref | template-ref]
```

(c) the `wait_async` directive

Figure 2: Syntax of the XMP directives for stencil communication (`[F]` is the line for XMP/-Fortran, and `[C]` the line for XMP/C. & indicates that the syntax rule continues.)

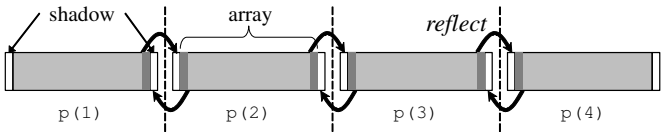


Figure 3: Workings of the `shadow` and `reflect` directives

Note that communications other than those induced by `reflect` can be asynchronous in XMP, and `wait_async` may have to handle them.

3 Omni XcalableMP

Omni XcalableMP is a reference implementation of an XMP compiler that is being developed as an open-source project by the HPCS Laboratory of the University of Tsukuba and Programming Environment Research Team of RIKEN AICS [1].

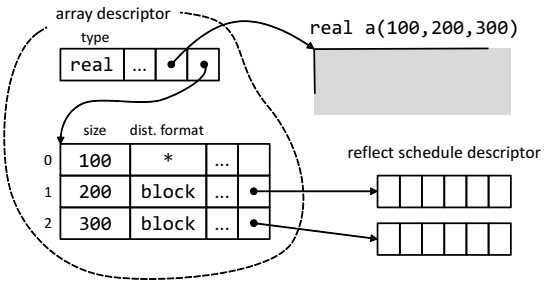


Figure 4: Descriptors in Omni XMP

Omni XMP consists of two major parts: a translator and a runtime library. The translator translates an XMP source program into a program that is in the base language and involves calls to the runtime routines. In particular, each executable directive, such as `reflect` and `wait_async`, in the source program is replaced with a sequence of runtime routine calls. The runtime library is in charge of, for example, parallel execution control, communication and synchronization, and memory management at runtime.

In the current implementation, the runtime library is based on MPI for portability, although those based on other communication libraries such as the extended RDMA interface of the K computer, which is dealt with in Section 5, and GASNet [3] are also being developed or planned.

The current implementation supports platforms of Linux clusters, Cray machines, the K computer, and any other machines on which MPI works.

4 Implementation

We implemented the `reflect` communication using two methods for general (MPI-supported) platforms: One method is based on MPI's derived datatype, and the other method is based on packing/unpacking buffers.

The XMP runtime system autonomously determines at runtime which of the two methods is used for stencil communications. In addition, users can explicitly specify the method with an environment variable.

4.1 Reflect Schedule Descriptor

The Omni XMP runtime system manages a descriptor of each distributed array to be referenced as necessary by the runtime library. The lifetime of the descriptor is the same as that of the corresponding array. In addition to this array descriptor, if a shadow area is declared for a dimension of an array, the runtime system creates a *reflect schedule descriptor (RSD)*, which stores information on the schedule of a `reflect` communication for the dimension, and links the RSD from the array descriptor (Figure 4). Once created, an RSD is reused repeatedly unless the schedule is changed by another `reflect` directive with different clauses specified. Table 1 shows the components of the RSD.

Table 1: Components of the reflect schedule descriptor

Type	Name	Description
`int`	`lo_width` `hi_width`	latest widths
`int`	`is_periodic`	latest periodic flag
`MPI_Datatype`	`datatype_lo` `datatype_hi`	MPI vector datatype
`MPI_Request`	`req[4]`	MPI request handles for upper/lower send/recv
`void*`	`lo_send_buf` `lo_recv_buf`	buffers for lower shadow
`void*`	`hi_send_buf` `hi_recv_buf`	buffers for upper shadow
`void*`	`lo_send_array` `lo_recv_array`	target positions in array for upper shadow
`void*`	`hi_send_array` `hi_recv_array`	target positions in array for upper shadow

`int`	`count`
	`blocklength`
	`stride`
		components of vector (used in pack/unpack)

`int`	`lo_rank` `hi_rank`	MPI ranks of neighboring nodes

4.2 Method 1: Derived Datatype

Any `reflect` communication can be performed as a point-to-point nonblocking communication of a message of type *vector* and length one, where vector is one of MPI's built-in derived datatypes consisting of equally spaced blocks and constructed by the function `MPI_TYPE_VECTOR`.

The vector datatype has three components: *count* for the number of blocks; *blocklength* for the number of elements in each block; and *stride* for the number of elements between start of each block.

The count, blocklength, and stride of the vector for `reflect` in the k'th dimension of an N-dimensional array are calculated as follows[2]:

$$\begin{aligned} count &= lsize_{k+1} \times \cdots \times lsize_{N-1} \\ blocklength &= lsize_0 \times \cdots \times lsize_{k-1} \times shadow_k \\ stride &= lsize_0 \times \cdots \times lsize_k \end{aligned}$$

where $lsize_i$ and $shadow_i$ represent the local size, which is the size of elements resident on each node, and the width of the lower or upper shadow area, in the i'th dimension of the array, respectively. Note that the local size includes the size of the shadow area.

The schedule of the nonblocking communication of the vector is bound to a *persistent communication request*, which is stored in the RSD and is used to initiate and complete persistent communication in functions `MPI_Startall` and `MPI_Waitall`, respectively (Figure 5).

Note that a schedule is created for each dimension of the array but, in the current implementation, persistent communications for all dimensions are issued asynchronously in a batch. This means that shadow areas at the corner boundaries of an array may not be updated properly, and, therefore, the nine-point difference cannot be handled. This problem can be resolved

[2]This applies to the Fortran-style column-major ordering of array elements. These calculations for C can be obtained easily but are not presented herein

```
1  // create datatypes
2  for (i = 0; i < ndims; i++){
3   MPI_Type_vector(count, blocklength*lwidth, stride, MPI_BYTE, &reflect->dt_lo);
4   MPI_Type_commit(&reflect->dt_lo);
5   ...
6  }
7
8  // initiate persistent comms.
9  for (i = 0; i < ndims; i++){
10  MPI_Recv_init(rbuf_lo, 1, reflect->dt_lo, src, tag, comm, &reflect->req[0]);
11  ...
12  MPI_Send_init(sbuf_hi, 1, reflect->dt_hi, dst, tag, comm, &reflect->req[3]);
13 }
14
15 // do persistent comms.
16 MPI_Startall(4*ndims, reflect->req);
17 MPI_Waitall(4*ndims, reflect->req, status);
```

Figure 5: Overview of derived-datatype method

easily by issuing persistent communications for each dimension synchronously and in sequence. However, for asynchronous `reflect` (described in Section 4.4), communications between ordinal neighbor nodes should be implemented in order to properly update the shadow area. The pack/unpack and the RDMA methods described in the following sections also have the same problem.

4.3 Method 2: Pack/Unpack

The method of communication of a message of type vector performed by the MPI library is implementation-dependent. One implementation can pack a vector into a contiguous buffer before sending data, whereas another implementation might send blocks of a vector one by one without packing. In general, internal packing/unpacking in sending/receiving a vector should be considered to be neither fully optimized nor multithreaded even in a multicore environment. Note that it is theoretically possible to parallelize packing/unpacking vectors, whereas this is not possible for a general datatype.

In order to achieve higher performance primarily in multicore environments, routines for packing/unpacking vectors are multithreaded using an OpenMP directive. Note that the specification states that an XMP directive is single-threaded and therefore an implementation can use multithreading to parallelize the corresponding runtime library routines.

However, such parallelization is effective only when more than one processor core is available in an XMP node (i.e., when using hybrid parallelization). Therefore the Omni XMP runtime system determines whether the packing/unpacking operation is to be executed in parallel, using an OpenMP API runtime library routine `omp_get_num_procs`, which returns the number of processors (cores) available to the program.

Figure 7 shows the internal packing routine `_XMPF_pack_vector` in Omni XMP, where variables `count`, `blocklength`, and `stride` are the same as those of the derived-datatype method. The loop is executed in parallel only if the number of available processor cores is greater than one and the amount of packing/unpacking operation is large enough for parallelization. The `THRESHOLD` variable indicates the threshold of the amount for parallelization, and the appropriate value of `THRESHOLD` depends on the environment.

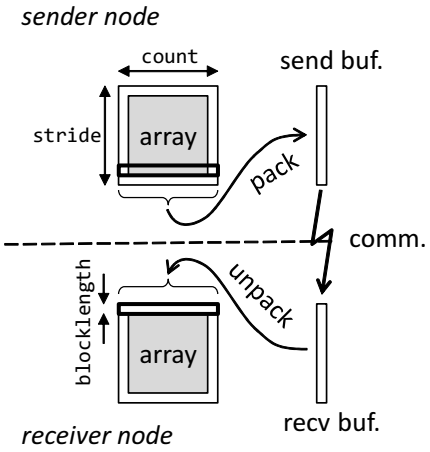


Figure 6: Packing/Unpacking a vector in `reflect`

```
1  void _XMPF_pack_vector(char * restrict dst, char * restrict src,
2                         int count, int blocklength, int stride){
3
4    if (_xmp_omp_num_procs > 1 && count * blocklength > THRESHOLD){
5  #pragma omp parallel for
6      for (int i = 0; i < count; i++){
7        memcpy(dst + i * blocklength, src + i * stride, blocklength);
8      }
9    }
10   else {
11     for (int i = 0; i < count; i++){
12       memcpy(dst + i * blocklength, src + i * stride, blocklength);
13     }
14   }
15
16 }
```

Figure 7: Packing routine

The communication buffer used for packing/unpacking in this method is managed by the runtime system. In the current implementation, once allocated for the dimension of an array, the communication buffer persists and is reused repeatedly for the lifetime of the array.

4.4 Asynchronous Communication

As shown in Section 2.3.2, a `reflect` communication can be asynchronous when the `async` clause specified in a `reflect` directive.

Such an "asynchronous `reflect`" is handled by the Omni XMP runtime system through MPI request handles associated with the nonblocking communications issued for it. The asynchronous `reflect` proceeds at runtime as follows:

1. At a `reflect` directive, a set of nonblocking communications is issued, and their request handles are stored in the *asynchronous communication table (ACT)*, which is a hash table

with *async-id*s as the hash keys.
2. The communications proceed while possibly overlapping some computations.
3. At an `wait_async` directive, the ACT is retrieved with the specified *async-id* to obtain the corresponding request handle, and issues `MPI_Waitall` to complete the nonblocking communications associated with the request.

Note that the `wait_async` directive is used to complete asynchronous communications other than `reflect`, and, therefore, the above mechanism is designed to be applicable to any asynchronous communications in XMP.

In the current implementation, asynchronous `reflect` is performed in the derived-datatype method described in Section 4.2, because issuing a nonblocking communication as early as possible without packing/unpacking in order to facilitate overlapping the following computation is advantageous for achieving high performance.

5 RDMA-based Experimental Implementation

In this section, we present an experimental implementation of the `reflect` directive based on the extend RDMA interface of the K computer[3].

5.1 The Extended RDMA Interface

The MPI library of the K computer and FUJITSU's PRIMEHPC FX10 supercomputer provides users with the *extended RDMA interface*. The interface consists of a number of functions[4] that enable inter-node communication that makes the most of the underlying interconnect hardware, such as Network Interface Controllers (NICs).

When implementing `reflect` communications using RDMA writes of this interface, the following items must be considered.

- An array must be registered to the system and associated with a memory ID using the `FJMPI_Rdma_reg_mem` function, in advance of being accessed through this interface. In the current implementation, all of the distributed arrays with shadow are registered to be (possibly) accessed through the interface.
- The array must be distributed onto the entire node set that corresponds to MPI_COMM_WORLD, because the target process of RDMA is identified with the rank in MPI_COMM_WORLD.
- The availability for the RDMA writes, i.e., whether the shadow areas on the neighboring nodes are ready to be updated, must be explicitly confirmed by each node before issuing the RDMA writes, which means that synchronizations are needed before `reflect`.
- Completion of the RDMA writes, i.e., whether the shadow areas on the neighboring nodes have been updated, must be explicitly confirmed by each node using the `FJMPI_Rdma_poll_cq` function, which means that synchronizations are needed after `reflect`.
- A *tag* that is an integer from 0 to 14 can be assigned to an RDMA in order to identify the RDMA. Since an async-id is used as the tag in the asynchronous mode, the value of the async-id is restricted to the 0 to 14 range.

The third and fourth items are due to the collectiveness of the `reflect` communication.

[3]The implementation is experimental because this implementation has some limitations (e.g., the number of arrays having shadow areas) that are derived from those of the extend RDMA interface and has not yet been released.

[4]These functions are based on a low-level communication library dedicated to the K computer and FX10.

5.2 Method 3: RDMA

Normal Mode A normal `reflect` communication based on the extended RDMA interface is performed in the following steps.

1. Each node waits until all of the nodes reach this point (barrier synchronization);
2. issues an RDMA write for each block of the vector;
3. polls its NICs until all of the RDMA writes issued by the node are completed; and
4. waits until all of the nodes reach this point (barrier synchronization).

The first barrier synchronization guarantees that the neighboring nodes are available, and the second barrier synchronization guarantees that all of the actions involving the communication on both the local and remote nodes are completed.

The reason for the lack of packing/unpacking is that the latency of RDMA writes is sufficiently low and the overhead of issuing multiple RDMA writes is smaller than that of packing/unpacking buffers.

Asynchronous Mode Steps 1 and 2 above are performed by `reflect`, and steps 3 and 4 above by `wait_async`, with the following differences. At `reflect`, RDMA writes are issued while setting the async-id as a tag, and the number of RDMAs issued for the async-id is stored in ACT. At `wait_async`, the NIC is polled until as many RDMAs as extracted from ACT are completed.

6 Evaluation

Using XMP, we parallelized a prototype of the dynamical core of a climate model for large eddy simulation, SCALE-LES [19], which is a typical five-point stencil code in Fortran (Figure 8), and ran the prototype on the K computer [16] in order to evaluate the performance of each implementation of `reflect`. The performance of an MPI-based implementation was also evaluated for comparison. The language environment used was K-1.2.0-13. The problem dimensions were 512×512 horizontally and 128 vertically, and the execution time was measured for 500 time steps.

In this evaluation, we assigned one XMP node to one compute node of the K computer, where intra-node thread-level parallelism can be automatically extracted from node programs by the compiler. The condition for parallelizing packing/unpacking buffers in the pack/unpack method was that the count of a vector (`count` in Figure 7) was more than eight times greater than the number of available cores (`_xmp_omp_num_procs` in Figure 7), i.e., more than eight blocks per thread. Therefore, the length of each block (`blocklength` in Figure 7) was not considered in this evaluation.

For clarify, the evaluation results are presented in three graphs in Figure 9. For the purpose of comparison, some results are presented in more than one graph. The vertical axes in these graphs indicate the speedup of the execution time, relative to that on a single node, and the horizontal axes in these graphs indicate the number of nodes. The computation times of these implementations are approximately equal because their computation codes generated by Omni XMP are identical and are nearly equivalent to that of MPI . Therefore, the difference in the execution time comes from the difference in the communication time (Table 2).

Figure 9 (a) shows the performance of normal-mode `reflect` communications, where MPI indicates the results obtained for the hand-coded MPI version, XMP-dt indicates the results obtained for the derived-datatype method, and XMP-pack indicates the results obtained for

```
1  !$xmp nodes p(N1,N2)
2  !$xmp template t(IA,JA)
3  !$xmp distribute t(block,block) onto p
4    ...
5    real(8) :: dens(0:KA,IA,JA)
6    ...
7  !$xmp align (*,i,j) &
8  !$xmp&   with t(i,j) :: dens, ...
9  !$xmp shadow (0,2,2) :: dens, ...
10   ...
11 !$xmp reflect (dens, ...) width &
12 !$xmp&     (0,/periodic/2,/periodic/2)
13   ...
14 !$xmp loop (ix,jy) on t(ix,jy)
15         do jy = JS, JE
16          do ix = IS, IE
17           ...
18           do kz = KS+2, KE-2
19            ... dens(kz,ix+1,jy) + ...
20            ...
21           end do
22           ...
23          end do
24         end do
```

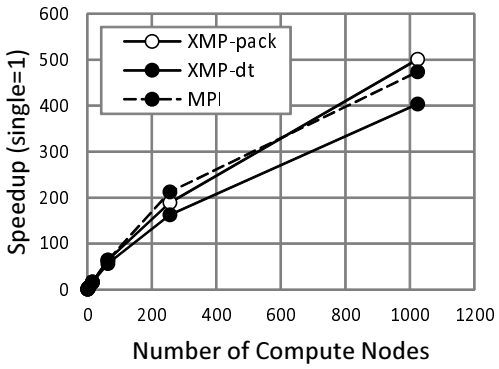
Figure 8: Code snippet of the target climate model

Table 2: Breakdown of the execution time (in seconds)

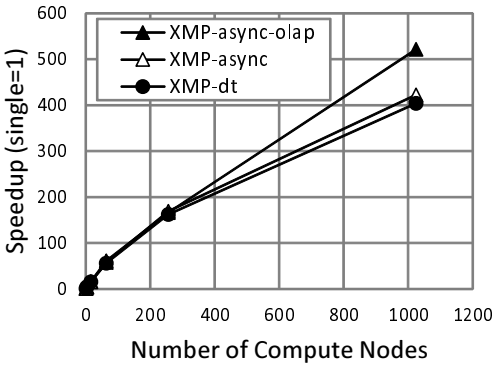
#nodes	4		16		64		256		1024	
	comm.	comp.	comm.	comp.	comm.	comp.	comm.	comp.	comm.	comp.
XMP-pack	8.98	413.1	7.09	102.3	4.95	23.3	4.28	5.35	2.46	1.17
XMP-dt	16.77	413.7	15.79	102.5	8.94	23.3	5.99	5.22	3.30	1.21
XMP-RDMA	7.19	415.4	7.04	101.0	4.80	23.4	4.06	5.22	2.79	1.12
XMP-async	29.50	416.9	15.47	103.3	8.35	23.2	5.48	5.29	3.05	1.26
MPI	15.39	423.6	8.82	100.0	5.47	23.0	3.61	4.98	4.16	N/A
MPI-RDMA	8.39	421.3	2.58	100.1	1.09	23.0	0.64	5.00	1.99	1.21

the pack/unpack method. The pack/unpack method is comparable in performance to the MPI version oand is faster than the MPI version for the 1,024-node execution. However, the results might depend on the fast inter-core hardware barrier of SPARC64 VIIIfx [24]. In fact, we observed that the pack/unpack method is not as effective for an average Linux cluster, as campared to the K computer. On the other hand, the derived-datatype method is slower than MPI. We verified that the derived-datatype method is faster than both the pack/unpack method and MPI in the flat-parallel environment. The results are not presented herein because of space limitations.

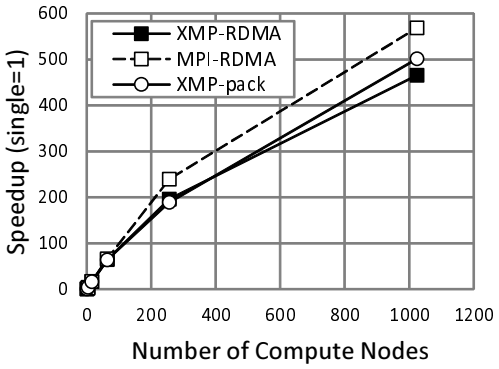
Figure 9 (b) shows the performance of asynchronous-mode `reflect` communications, where XMP-dt indicates the results obtained for the synchronous-mode derived-datatype method (for comparison), XMP-async indicates the results obtained for the asynchronous-mode `reflect` that do not overlap with the computations, and XMP-async-olap indicates the results obtained for as much part of the asynchronous-mode `reflect` as possible overlapped with the computations. The overhead introduced for asynchronous communication, such as management and



(a) normal `reflect`



(b) asynchronous `reflect`



(c) RDMA-based `reflect`

Figure 9: Evaluation results on the K computer

retrival of ACT, is not so large and the performance is improved significantly by overlapping communication with computation in the 1,024-node execution.

Figure 9 (c) shows the performance of RDMA-based `reflect` communications, where MPI-RDMA indicates the results obtained for the hand-coded RDMA-based version, XMP-pack indicates the results obtained for the pack/unpack method (not based on RDMA, for comparison), and XMP-RDMA indicates the results obtained for the RDMA-based method. The experimental implementation is slower than both the hand-coded RDMA-based implementation and the pack/unpack implementation because the barrier synchronizations before issuing RDMA writes and after completing RDMA writes are too strong to perform stencil communication efficiently. Actually, in the hand-coded implementation, point-to-point synchronizations between neighboring nodes are used instead of barrier synchronizations. In the future, synchronizations performed in RDMA-based `reflect` should be weakened in order to achieve higher performance.

7 Related Research

The `reflect` directive and its asynchronous mode of XMP originates from HPF/JA, which is an extension of High Performance Fortran for accelerating real-world applications [12, 20]. The function of partial reflection was first supported by HPF/SX V2 [17] and HPF/ES [23], the HPF compiler for NEC's SX-series supercomputers and the Earth Simulator, respectively, and later by the dHPF compiler developed by Rice University [6]. Since there is no specification for periodic stencil communication in either the HPF standard or the HPF/JA specification, to our knowledge, no compilers for HPF or HPF-like languages have supported periodic stencil communication yet. On the other hand, a region-based parallel language ZPL that supports periodic stencil communication has been reported [4].

The optimization of stencil communication in HPF is described in a previous study [13], in which a method of generating communications based on realignment was proposed and compile-time optimizations for multidimensional stencil communications were presented.

In [2, 11], implementations of mesh-based regular applications with coarrays, which is a one-sided communication feature from Co-Array Fortran or Fortran 2008, are compared with implementations of mesh-based regular applications with MPI, from the viewpoints of, for example, memory layout and the usage of communication buffers. Stencil communications based on coarrays were demonstrated to be effective in mesh-based regular applications and could, in some cases, outperform stencil communications based on MPI.

8 Conclusions and Future Research

We implemented three methods for stencil communication in the Omni XMP compiler. The first method based on derived-datatype messaging is simple and general, and could be efficient depending on the implementation of the underlying MPI library. The second method is based on packing/unpacking and has the advantage of being multithreaded in multicore environments. The third method, which is experimental and is based on the extended RDMA interface of the K computer, may be able to achieve higher performance, but at present has approximately the same performance as the second method because of exceedingly strong synchronizations.

Areas for future research include:

- managing `reflect` communications from/to ordinal neighbor nodes properly in nine-point difference stencil codes;
- setting an appropriate threshold for parallelizing packing/unpacking buffers in the pack-/unpack method;
- improving the performance of the RDMA-based method by reducing the strength of synchronizations; and
- providing a more portable and efficient implementation based on the one-sided communication of MPI-3.

Acknowledgements

The results were obtained in part using the K computer at the RIKEN Advanced Institute for Computational Science. The original code of the climate model used in the evaluation and its RDMA-based implementation were provided by Team SCALE of RIKEN AICS.

References

[1] Omni XcalableMP Compiler. `http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/xcalablemp/`.

[2] Richard Barrett. Co-array Fortran Experiences with Finite Differencing Methods. In *The 48th Cray User Group meeting, Lugano, Italy*, 2006.

[3] D. Bonachea. GASNet specification. Technical report, University of California, Berkeley (CSD-02-1207), October 2002.

[4] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, November 2001.

[5] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An Object–oriented Approach to Non–Uniform Cluster Computing. In *Proc. OOPSLA 05*, 2005.

[6] D. Chavarria-Miranda and J. Mellor-Crummey. An Evaluation of Data-Parallel Compiler Support for Line-Sweep Applications. *J. Instruction Level Parallelism*, 5, 2003.

[7] Collins, William D and Bitz, Cecilia M and Blackmon, Maurice L and Bonan, Gordon B and Bretherton, Christopher S and Carton, James A and Chang, Ping and Doney, Scott C and Hack, James J and Henderson, Thomas B and others. The Community Climate System Model Version 3 (CCSM3). *J. Climate*, 19(11):2122–2143, 2006.

[8] Cray Inc. Chapel Language Specification 0.93. `http://chapel.cray.com/spec/spec-0.93.pdf`, 2013.

[9] FUJITSU LIMITED. *Parallelnavi Technical Computing Language MPI User's Guide*, 2013.

[10] Furumura, Takashi and Chen, Li. Parallel simulation of strong ground motions during recent and historical damaging earthquakes in Tokyo, Japan. *Parallel Computing*, 31(2):149–165, 2005.

[11] Manuel Hasert, Harald Klimach, and Sabine Roller. CAF versus MPI - Applicability of Coarray Fortran to a Flow Solver. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 228–236, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] Japan Association of High Performance Fortran. HPF/JA Language Specification. `http://www.hpfpc.org/jahpf/spec/hpfja-v10-eng.pdf`, 1999.

[13] Tsunehiko Kamachi, Kazuhiro Kusano, Kenji Suehiro, and Yoshiki Seo. Generating Realignment-Based Communication for HPF Programs. In *Proc. IPPS*, pages 364–371, 1996.

[14] Ken Kennedy, Charles Koelbel, and Hans Zima. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Proc. 3rd ACM SIGPLAN History of Programming Languages Conf. (HOPL-III)*, pages 7–1–7–22, San Diego, California, June 2007.

[15] Message Passing Interface Forum. MPI: A Message Passing Interface Standard Version 3.0. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, 2012.

[16] Hiroyuki Miyazaki, Yoshihiro Kusano, Naoki Shinjou, Fumiyoshi Shoji, Mitsuo Yokokawa, and Tadashi Watanabe. Overview of the K computer. *FUJITSU Sci. Tech. J.*, 48(3):255–265, 2012.

[17] Hitoshi Murai, Takuya Araki, Yasuharu Hayashi, Kenji Suehiro, and Yoshiki Seo. Implementation and Evaluation of HPF/SX V2. *Concurrency and Computation — Practice & Experience*, 14(8–9):603–629, 2002.

[18] Robert W. Numrich and John Reid. Co-arrays in the next Fortran Standard. *ACM Fortran Forum*, 24(2):4–17, 2005.

[19] Team SCALE. SCALE-LES. `http://scale.aics.riken.jp/scale-les/`.

[20] Yoshiki Seo, Hidetoshi Iwashita, Hiroshi Ohta, and Hitoshi Sakagami. HPF/JA: extensions of High Performance Fortran for accelerating real-world applications. *Concurrency and Computation — Practice & Experience*, 14(8–9):555–573, 2002.

[21] UPC Consortium. UPC Specifications, v1.2. Technical report, Lawrence Berkeley National Lab

(LBNL-59208), 2005.

[22] XcalableMP Specification Working Group. XcalableMP Specification Version 1.1. `http://www.xcalablemp.org/xmp-spec-1.1.pdf`, 2012.

[23] Takashi Yanagawa and Kenji Suehiro. Software System of the Earth Simulator. *Parallel Computing*, 30(12):1315–1327, 2004.

[24] Toshio Yoshida, Mikio Hondo, and Ryuji Kan Go Sugizaki. SPARC64 VIIIfx: CPU for the K computer. *FUJITSU Sci. Tech. J.*, 48(3):274–279, 2012.

Productivity and Performance of the HPC Challenge Benchmarks with the XcalableMP PGAS Language

Masahiro Nakao[1,2], Hitoshi Murai[2], Takenori Shimosaka[2] and Mitsuhisa Sato[1,2]

[1] Center for Computational Sciences, University of Tsukuba, Japan
mnakao@ccs.tsukuba.ac.jp
[2] RIKEN Advanced Institute for Computational Science, Japan

Abstract

The present paper introduces designs of the XcalableMP PGAS language for improved productivity and performance of a high performance computing (HPC) system. The design of a unique XcalableMP programming model is based on both local-view and global-view models. This design allows programmers to easily develop HPC applications. Moreover, in order to tune HPC applications, XcalableMP provides inquiry functions for programmers to obtain local memory information of a global array. In the present paper, we evaluate the productivity and the performance of XcalableMP through implementations of the High-Performance Computing Challenge (HPCC) Benchmarks. We describe the implementations of three HPCC benchmarks, including RandomAccess, High-performance Linpack (HPL), and Fast Fourier Transform (FFT). In order to evaluate the performance of XcalableMP, we used the K computer, which is a leadership-class HPC system. As a result, we achieved 163 GUPS with RandomAccess (using 131,072 CPU cores), 543 TFlops with HPL (using 65,536 CPU cores), and 24 TFlops with FFT (using 262,144 CPU cores). These results reveal that XcalableMP has good performance on the K computer.

1 Introduction

At present, leadership-class high-performance computing (HPC) systems consist of hundreds of thousands of compute nodes, and the number of the compute nodes will continue to increase. Applications run on the system are used so many times that programmers require a high-performance programming language to reduce the execution time. Moreover, in order to reduce the programming and maintenance costs, programmers also require the programming language to be productive. Partitioned Global Address Space (PGAS) model languages, such as SHMEM[12], Global Arrays[23], Coarray Fortran[24], Titanium[29], Unified Parallel C[13], Chapel[11], and X10[25], are emerging as alternatives to Message Passing Interface (MPI)[26]. In particular, the greater part of Coarray Fortran is incorporated into the Fortran 2008 standard. In addition to these languages, we have been designing and developing a new PGAS model language called XcalableMP (XMP) [19, 21, 22, 28], which is a directive-based language extension of C and Fortran.

Since most existing HPC applications are written in MPI, PGAS languages are not widely used. One reason for this is that the productivity and performance of PGAS languages on an HPC system remain unclear. Therefore, in the present study, we reveal the productivity and performance of the XMP PGAS language using a maximum of 32,768 compute nodes (262,144 CPU cores) of the K computer. The K computer was ranked 4th in the Top500[9] on June, 2013. In order to measure the productivity and performance, we have used the High-Performance Computing Challenge (HPCC) benchmarks[5]. The HPCC benchmarks are a set

of benchmarks to evaluate multiple attributes, such as the random access speed of memory, system interconnect, and the processor on an HPC system. The HPCC benchmarks are also used at the HPCC Award Competition, which has been held at the supercomputer conference since 2005. The HPCC Award Competition consists of two classes. The class 1 competition uses four of the HPCC benchmarks, namely, RandomAccess, High-performance Linpack (HPL), Fast Fourier Transform (FFT), and STREAM, to evaluate the overall performance of an HPC system. These benchmarks are frequently used in scientific fields. The class 2 competition uses three or four of the HPCC benchmarks to evaluate the productivity and performance of a programming language. In the present paper, we have implemented RandomAccess, HPL, and FFT benchmarks because parallelizing the STREAM benchmark is easy. Through these implementations and evaluations, we investigate potential improvements in HPC systems that may be achieved through proper XMP design.

To this end, the present paper makes the following specific contributions: (1) Designs of the XMP programming model for an HPC system are proposed and their effectiveness is validated; and (2) The implementation of the HPCC benchmarks using XMP and the tuning of these benchmarks are presented. These contributions are also useful for other PGAS languages.

The remainder of the present paper is structured as follows. Section 2 presents the requirements of a promising programming model on an HPC system. Section 3 introduces an overview of XMP. Section 4 describes implementations and evaluations of the HPCC benchmarks. Section 5 discusses the productivity and performance of the HPCC benchmarks in XMP. Section 6 summarizes the present paper and describes areas for future research.

2 Requirements of a Programming Model on High-performance Computing System

A promising programming model for HPC systems must have both high productivity and high performance. In this section, we consider the features necessary for a promising programming model.

2.1 High Productivity

2.1.1 Components of Productivity

The productivity of HPC applications is based on not only the programming cost but also the porting, tuning, maintenance, and educational costs. In addition, the productivity is related to the ability to easily reuse and extend a part of the application. Therefore, the productivity should not be evaluated based solely on the Source Lines Of Code (SLOC), which should be one of indicators used in the evaluation of the productivity.

2.1.2 Performance Portability

In general, the lifetime of an HPC application is longer than the lifetime of the hardware. Once the HPC application has been created, the application is used for several decades while maintaining, expanding, and changing its data structure and/or algorithm. Therefore, the HPC application should enable high performance on different machines with little effort. In order to meet this requirement, the source code must clearly indicate the behavior of the computer, for example, access to local memory and data communication.

2.1.3 Usage of Numerical Libraries

In order to reduce the porting and programming costs and allow high performance, HPC applications often use numerical libraries such as BLAS[1] and Scalapack[2]. Therefore, a promising language must be able to use the numerical libraries directly or to use a language-specific library such as UPCBLAS[16]. In order to use numerical libraries directly from the PGAS model language, the language should expose local memory information of its global address space.

2.1.4 Coexistence with MPI

At present, most HPC applications are written in MPI. Since the performances of existing MPI applications are extremely good, it is not worth rewriting all of the existing MPI applications in terms of a promising language. Therefore, the promising language and MPI should coexist. In particular, the promising language must be able to call an MPI library and an MPI application. Moreover, a library and an application written in the promising language must be able to be called from an MPI application. In order to meet these requirements, the promising language must deal with MPI objects such as an MPI communicator.

2.2 High Performance

2.2.1 Ease of Tuning

In order to tune high-performance applications, programmers must perform hardware-aware programming. In other words, each line of the promising language allows programmers to clearly understand what happens on the hardware.

2.2.2 Support of Any Parallelizations

In order to increase the performance/cost ratio, the commonly used HPC system is a multicore cluster and may also have accelerators such as a Graphics Processing Unit (GPU) or a Many Integrated Core (MIC). Therefore, the promising language should control hybrid-parallelization, which combines process-parallelization with thread-parallelization, and parallelization on accelerators. At present, MPI and OpenMP are frequently used for hybrid parallelization, and CUDA and OpenACC[7] are frequently used for parallelization on accelerators. Note that the promising language does not need to support all parallelizations as its function. For example, if the promising language can use OpenMP directives, the promising language does not have to support thread-parallelization as its function.

3 Overview of XcalableMP

In this section, we describe some of the features of the design of XMP related to performance and productivity, as described in Section 2. In addition, we introduce the implementation of an XMP compiler.

3.1 Design

The XMP specification[28] has been designed by the XMP Specification Working Group, which consists of members from academia, research laboratories, and industries. This Working Group is supported by the PC Cluster Consortium in Japan[8].

XMP/Fortran

integer function	xmp_array_gtol(*d*, *g_idx*, *l_idx*)
type(xmp_desc)	*d*
integer	*g_idx(NDIMS)*
integer	*l_idx(NDIMS)*

XMP/C

void	xmp_array_gtol(xmp_desc_t *d*, int *g_idx[]*, int *l_idx[]*)

Figure 1: Example of XMP Inquiry Functions

```
 1  #include "mpi.h"
 2  #include "xmp.h"
 3  #pragma xmp nodes p(4)
 4
 5  void main(int argc, char *argv[]) {
 6    xmp_init_mpi(argc, argv);
 7    MPI_Comm comm = xmp_get_mpi_comm();
 8
 9    user_mpi_func(comm);
10
11    xmp_finalize_mpi();
12  }
```

Figure 2: Example of the Use of the MPI Programming Interface in XMP/C

3.1.1 Familiar HPC Language (related to Section 2.1.1)

In order to reduce programming and educational costs, XMP is extended to familiar HPC languages, such as Fortran and C, and these extensions are referred to herein as XMP/Fortran and XMP/C, respectively.

3.1.2 Support of Global-view and Local-view (related to Section 2.1.1)

In order to develop various applications, the XMP programming model is an SPMD under the global-view and local-view models. In the global-view model, XMP enables parallelization of an original sequential code using minimal modification with simple directives such as OpenMP. In the local-view model, the programmer can use coarray syntax in both XMP/Fortran and XMP/C. In particular, XMP/Fortran is designed to be compatible with Coarray Fortran.

3.1.3 XMP Inquiry Functions (related to Sections 2.1.2, 2.1.3, and 2.2.1)

In order to use numerical libraries, XMP inquiry functions provide local memory information of a global array defined in the XMP global-view model. Figure 1 shows one of the XMP inquiry functions `xmp_array_gtol()`. This function translates an index (specified by *g_idx*) of a global array (specified by descriptor *d*) into the corresponding index of its local section and sets to an array (specified by *l_idx*). In addition to this function, the XMP specification defines inquiry functions that enable the collection of other local memory information, such as the pointer, size, and leading dimension of a global array. The XMP inquiry functions enable the use of numerical libraries and tune the application using the local memory information.

3.1.4 MPI Programming Interface (related to Section 2.1.4)

In order to call an MPI program from an XMP program, the MPI programming interface, a function `xmp_get_mpi_comm()` in XMP/C, is provided. This function returns a handle of an MPI communicator associated with the execution of processes. In addition to this function, the XMP specification defines two functions, `xmp_init_mpi()` and `xmp_finalize_mpi()`, to initialize and finalize the MPI execution environment in XMP/C. Figure 2 shows an example of using these functions. Line 2 includes xmp.h, in which the functions are defined. Line 3 defines the XMP node set, which is a process unit and typically corresponds to an MPI process. Thus, this program is executed on four nodes. Lines 6 and 11 perform initialization and finalization of the MPI execution environment. Line 7 obtains an MPI communicator, and line 9 sends the MPI

communicator to the user-defined MPI function. Similarly, XMP/Fortran also provides these functions.

The XMP specification has not yet provided a function of how to call an XMP program from an MPI program. However, in the implementation of FFT using XMP (described in Section 4.3.1), an FFT main kernel is called by an MPI program (Figure 13) because an Omni XMP compiler informally supports the function. Now, we are planning to design a new XMP specification to support the function.

3.1.5 Leverage HPF's Experience (related to Section 2.2.1)

Part of the design of XMP is also based on High Performance Fortran (HPF)[17, 18]. Thus, some concepts of HPF, such as the use of a template, which is a virtual global index, have been inherited. However, all communication and synchronization actions occurred only at points of the XMP directive or coarray syntax because of performance awareness. In other words, an XMP compiler does not automatically insert communication calls. This design is different from HPF. Since it is clear what each line does in XMP, this design enables programmers to easily tune an application.

3.1.6 Use of Other Parallelizations (related to Section 2.2.2)

In XMP, OpenMP and OpenACC pragmas can also be used only for local calculations because programmers can obtain the local memory information (described in Section 3.1.3). We have been designing a new XMP specification for mixing communication with these pragmas.

On the other hand, XMP-dev[19, 20], which is an extended XMP, has been proposed. Since XMP-dev provides pragmas for accelerators, XMP-dev allows programmers to easily develop parallel applications on an accelerator cluster.

3.2 Implementation

We have been developing an Omni XMP compiler[6, 19] as a prototype compiler to compile XMP/C and XMP/Fortran codes. The Omni XMP compiler is a source-to-source compiler that translates an XMP/C or XMP/Fortran code into a parallel code using an XMP runtime library. The parallel code is compiled by the native compiler of the machine (e.g., Intel compiler).

The latest Omni XMP compiler (version 0.6.2-beta) has been optimized for the K computer. For example, in order to use high-speed one-sided communication on the K computer, the coarray syntax is translated into calling the extended RDMA interface provided by the K computer [15]. In Section 4, we use the Omni XMP Compiler to evaluate the HPCC benchmarks.

4 High-Performance Computing Challenge Benchmarks in XcalableMP

In this section, we introduce the proposed implementations of the HPCC benchmarks and describe the evaluations thereof. In order to evaluate the performance of these implementations, we used a maximum of 32,768 compute nodes (262,144 CPU cores) of the K computer (CPU: SPARC64 VIIIfx 2.0 GHz (eight CPU cores), Memory: DDR3 SDRAM 16 GB 64 GB/s, Network: Torus fusion six-dimensional mesh/torus network 5GB/s x2 (bi-directional) x10 (links per node)). In addition, in Section 4.4, we evaluate the performance of these implementations on a general PC cluster.

An overview of each benchmark is presented below.

- The RandomAccess benchmark measures the performance of random integer updates of memory via interconnect.
- The HPL benchmark measures the floating point rate of execution to solve a dense system of linear equations using LU factorization.
- The FFT benchmark measures the floating point rate of execution for double-precision complex one-dimensional Discrete Fourier Transform.

4.1 RandomAccess

4.1.1 Implementation

The proposed algorithm is iterated over sets of CHUNK updates on each node. In each iteration, the proposed algorithm calculates for each update the destination node that owns the array element to be updated and communicates the data with each node. This communication pattern is known as complete exchange or all-to-all personalized communication, which can be performed efficiently by an algorithm referred to as the recursive exchange algorithm when the number of nodes is a power of two [14].

We implemented an algorithm with a set of remote writes to a coarray in local-view programming using XMP/C. Note that the number of the remote writes is also sent as an additional first element of the data. A point-to-point synchronization is specified with the XMP's **post** and **wait** directives in order to realize asynchronous behavior of the algorithm.

Figure 3 shows part of the proposed RandomAccess code. Line 1 declares arrays *recv[][]* and *send[][]* as coarrays. Since these arrays on both hand sides of a coarray operation should be coarrays due to a restriction of the interface of the K computer, the array *send[][]* on the right-hand side, which was originally a normal (non-coarray) data, is declared as a coarray. In line 18, the variable *nsend*, which is the number of transfer elements, is set to the first element of array *send[][]* to be used by the destination node to update its local table. In line 19, XMP/C extends the syntax of the array reference of the C language so that the "array section notation" can be specified instead of an index. The number before the colon in square brackets (*0*) indicates the start index of the section to be accessed, and the number after the colon (*nsend+1*) indicates its length. The number in square brackets after an array and the colon (*ipartner*) indicates the node number. Thus, line 19 means that elements from *send[isend][0]* to *send[isend][nsend]* are put to those from *recv[j][0]* to *recv[j][nsend]* in the *ipartner* node. In line 20, the **sync memory** directive is used to ensure the remote definition of a coarray is complete. In line 21 and 27, the **post** and **wait** directives are used for point-to-point synchronization. The **post** directive sends a signal to the node *ipartner* to inform that the remote definition for it is completed. Each node waits at the **wait** directive until receiving the signal from the node *jpartner*.

4.1.2 Performance

We performed the proposed implementation, referred to as flat-MPI, of RandomAccess on each CPU core. The table size is equal to 1/4 of the system memory. Figure 4 shows the performance results. For comparison, we evaluated the modified hpcc-1.4.2[5] RandomAccess, for which the functions for sorting and updating the table are specifically optimized for the K computer. The best performance of the XMP implementation is 163 GUPS (Giga UPdates per Second) in 131,072 CPU cores. Figure 4 shows that the XMP implementation and modified hpcc-1.4.2 have approximately the same performances.

```
1 u64Int recv[MAXLOGPROCS][RCHUNK+1]:[*],
          send[2][CHUNKBIG+1]:[*];       // Declare Coarrays
2 ...
3 for (j = 0; j < logNumProcs; j++) {
4   nkeep = nsend = 0;
5   isend = j % 2;
6   ipartner = (1 << j) ^ MyProc;
7   if (ipartner > MyProc) {
8     sort_data(data, data, &send[isend][1], nkept, &nsend, ...);
9     if (j > 0) {
10      jpartner = (1 << (j-1)) ^ MyProc;
11 #pragma xmp wait(p(jpartner+1))
12 #pragma xmp sync_memory
13      nrecv = recv[j-1][0];
14      sort_data(&recv[j-1][1], data, &send[isend][1], nrecv, &nsend, ...);
15    }
16  }
17  else {   ...   }
18  send[isend][0] = nsend;
19  recv[j][0:nsend+1]:[ipartner+1] = send[isend][0:nsend+1];
20 #pragma xmp sync_memory
21 #pragma xmp post(p(ipartner+1), 0)
22  if (j == (logNumProcs - 1)) update_table(data, Table, nkeep, ...);
23  nkept = nkeep;
24 }
25 ...
26 jpartner = (1 << (logNumProcs-1)) ^ MyProc;
27 #pragma xmp wait(p(jpartner+1))
28 #pragma xmp sync_memory
29 nrecv = recv[logNumProcs-1][0];
30 update_table(&recv[logNumProcs-1][1], Table, nrecv, ...);
```

Figure 3: Source Code of RandomAccess

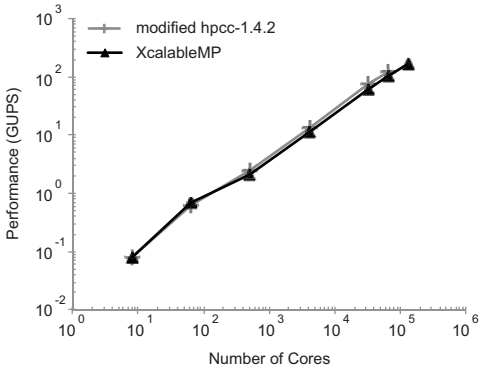


Figure 4: Performance of RandomAccess on the K Computer

```
1 double A[N][N];
2 #pragma xmp template t(0:N-1, 0:N-1)
3 #pragma xmp nodes p(P,Q)
4 #pragma xmp distribute t(cyclic(NB), cyclic(NB)) onto p
5 #pragma xmp align A[i][j] with t(j,i)
```

Figure 5: Define the Distribution Array for High-performance Linpack

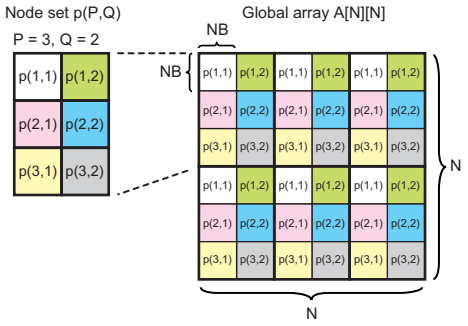


Figure 6: Block-cyclic Distribution in High-performance Linpack

4.2 High-performance Linpack

4.2.1 Implementation

Figure 5 shows part of the proposed HPL implementation in XMP/C, where each node distributes a global array *A[][]* in a block-cyclic manner, as is the case with hpcc-1.4.2 HPL. In this code, a **template** directive declares a two-dimensional template *t*, and a **node** directive declares a two-dimensional node set *p*. A **distribute** directive distributes the template *t* onto $P \times Q$ nodes in the same block size (where *NB* is the block size). Finally, an **align** directive declares a global array *A[][]* and aligns *A[][]* with the template *t*. Figure 6 shows the block-cyclic

```
1 double L[N][NB];
2 #pragma xmp align L[i][*] with t(*,i)
3 :
4 #pragma xmp gmove
5 L[j+NB:N-j-NB][0:NB] = A[j+NB:N-j-NB][j:NB];
```

Figure 7: Source Code of a **gmove** Directive in High-performance Linpack

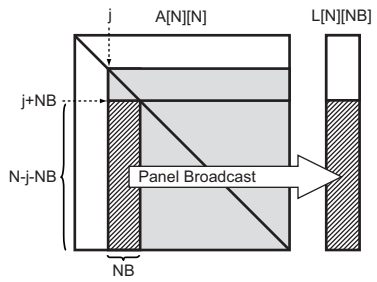


Figure 8: Panel Broadcast Using a **gmove** Directive in Figure 7

```
1  #include "xmp.h"
2  int g_idx[2], l_dx[2];
3  ...
4  // set g_idx
5  ...
6  xmp_desc_t A_desc = xmp_desc_of(A); // A is a global array
7  xmp_array_gtol(A_desc, g_idx, l_idx);
8  int local_y = l_idx[0];
9  int local_x = l_idx[1];
10 cblas_dgemm(..., N/Q-local_y, N/P-local_x, ...,
           &L[g_idx[0]][0], ..., &A[g_idx[0]][g_idx[1]], ...);
```

Figure 9: Example of a Calling BLAS Library

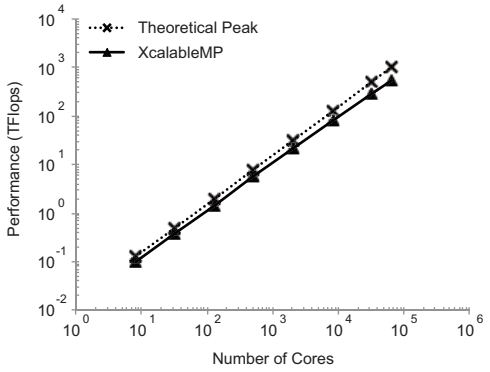


Figure 10: Performance of High-performance Linpack on the K Computer

distribution in Figure 5.

Figures 7 and 8 indicate a panel broadcast operation in HPL using the **gmove** directive and array section notation. The array *L[][]* is also distributed in the block-cyclic manner, but only the first dimension of the array *L[][]* is distributed. Thus, target elements of the array *A[j+NB:N-j-NB][j:NB]* (stripe block in Figure 8) are broadcast to the array *L[j+NB:N-j-NB][0:NB]* that exists on each node. Note that since a non-blocking **gmove** directive has not yet been implemented in the Omni XMP compiler, this **gmove** directive is a blocking operation.

In HPL, the performance of matrix multiply is significant important. Thus, high-performance *DGEMM()* and *DTRSM()* functions in the BLAS library should be used. In order to use the BLAS library optimized for the K computer, we call *DGEMM()* and *DTRSM()* functions using XMP inquiry functions (refer to Section 3.1.3). Figure 9 shows part of a code for calling the BLAS library. Line 1 includes the header file xmp.h to use XMP inquiry functions. Line 6 gets the descriptor of the global array *A[][]*, and Line 7 sets the local indices from global indices. Line 10 calls the DGEMM function using local indices. The *N/Q-local_y* and *N/P-local_x* are the local widths of the column and row of the calculated matrixes. In addition, XMP has a rule that a pointer of a global array indicates a local pointer on the node to which it is distributed. Thus, *&L[g_idx[0]][0]* and *&A[g_idx[0]][g_idx[1]]* indicate proper local pointers. In contrast to the hpcc-1.4.2 HPL, the *DGEMM()* function is called only one time in each update operation, because a node has all of the data for the matrix multiply.

```
 1 COMPLEX*16 :: A(NX,NY), A_WORK(NX,NY), B(NY,NX)
 2 !$XMP template ty(NY)
 3 !$XMP template tx(NX)
 4 !$XMP nodes p(*)
 5 !$XMP distribute ty(block) onto p
 6 !$XMP distribute tx(block) onto p
 7 !$XMP align A(*,i) with ty(i)
 8 !$XMP align A_WORK(i,*) with tx(i)
 9 !$XMP align B(*,i) with tx(i)
10   ...
11 !$XMP gmove
12 A_WORK(1:NX,1:NY) = A(1:NX,1:NY)
13
14 !$XMP loop (i) on tx(i)
15 !$OMP parallel do
16 DO 70 I=1,NX
17   DO 60 J=1,NY
18     B(J,I)=A_WORK(I,J)
19   60 CONTINUE
20 70 CONTINUE
```

Figure 11: Source Code of the Fast Fourier Transform

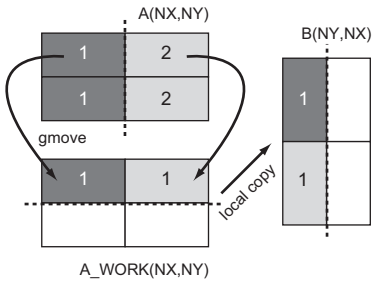


Figure 12: Matrix Transpose in the Fast Fourier Transform

4.2.2 Performance

We performed the proposed implementation with eight threads per process on one node. The size of the array *A[][][]* is approximately 90% of the system memory. Figure 10 shows the performance and the theoretical peak performance of the system. The best performance is 543 TFlops in 65,536 CPU cores. This performance is approximately 53% of the theoretical peak. The performance of using only eight CPU cores (on one node) is 100 GFlops, which is approximately 78% of the theoretical peak. The parallelization efficiency appears not to be very good. The main reason is that an effective panel broadcast operation has not yet been implemented. We consider this issue in Section 5.1.2.

4.3 Fast Fourier Transform

4.3.1 Implementation

We parallelized a subroutine, "PZFFT1D0", which is the main kernel of the FFT. This subroutine is included in ffte-5.0 [3]. Figure 11 shows part of the proposed FFT implementation in XMP/Fortran. In lines 2 through 9, the **template**, **nodes**, **distribute**, and **align** directives describe the distribution of arrays *A()*, *A_WORK()*, and *B()* in a block manner.

In a six-step FFT, which is an algorithm used in ffte-5.0, a matrix transpose operation must be performed before one-dimensional FFT. In the sequential version FFT, the matrix transpose is implemented by local memory copy between *A()* and *B()*. In the XMP version, the matrix transpose operation is implemented by the **gmove** directive and local memory copy in lines 11 through 20. Figure 12 shows how to transpose matrix *A()* to *B()*. In figure 12, the number is the node number to which the block is allocated, and the dotted lines indicate how to distribute the matrices. Since the distribution manner of the arrays (*A()* and *B()*) is different, node 1 does not have all of the elements of matrix *A()*, which are needed for the transpose. The **gmove** directive is used to collect these elements. Initially, a new array *A_WORK()* is declared to store the elements. *A_WORK()* is distributed by template *tx*, which is used to distribute *B()*. Consequently, the local block of *A_WORK()* and that of *B()* have the same shape. By the all-to-all communication of the **gmove** directive in lines 11 and 12, all elements needed

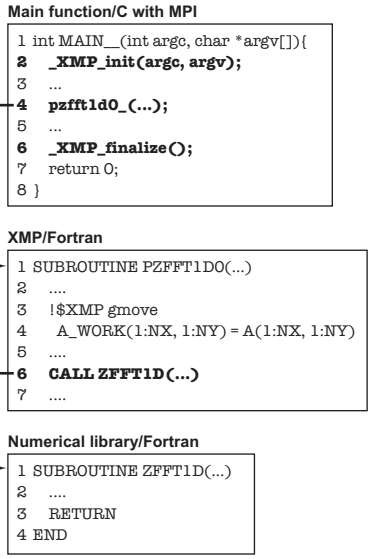


Figure 13: Example of Calling an XMP Program from an MPI Program

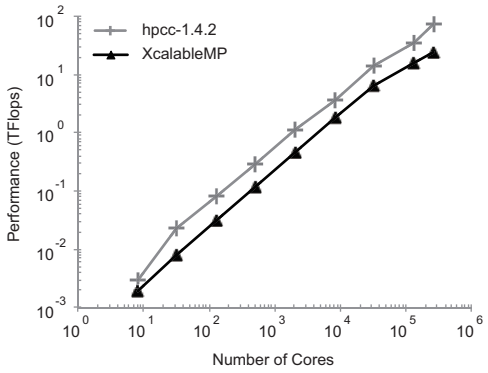


Figure 14: Performance of the Fast Fourier Transform on the K computer

for transpose are stored in *A_WORK()*. Finally, lines 14 through 20 copy the elements to *B()* using the **loop** directive and OpenMP thread-parallelization. The **loop** directive distributes iterations that are specified by the on clause and are executed in parallel.

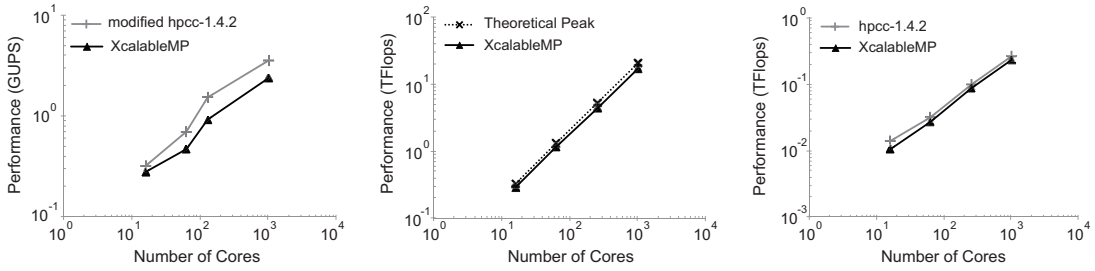
This FFT implementation is a good demonstration of how to mix an XMP program with an MPI program. The subroutine "PZFFT1D0" written in XMP/Fortran is called by the main function written in the C language with MPI. In addition, the "PZFFT1D0" calls other subroutines in ffte-5.0. Figure 13 shows these procedures, which are the same as the hpcc-1.4.2 FFT. Lines 2 and 6 in "Main function/C" of Figure 13 call the `_XMP_init()` and `_XMP_finalize()` functions to initialize and finalize an XMP execution environment. These functions are used internally in the Omni XMP compiler and also set up an MPI execution environment. Since this method is informal, we intend to design official functions for how to call an XMP program from another language.

4.3.2 Performance

We performed the proposed implementation with eight threads per process on one node. For comparison, we evaluated the hpcc-1.4.2 FFT. The program size is approximately 60% of the system memory. Figure 14 shows the performance of the implementations. The best performance of the XMP implementation is 24 TFlops for 262,144 CPU cores. The performance of the XMP implementation is approximately half that of the hpcc-1.4.2 FFT. The primary reason for this difference is that an internal packing operation for arrays is slow when using the **gmove** operation. We consider this issue in Section 5.1.3.

4.4 Performance on a General PC Cluster

In this section, we describe the performance of the proposed implementations on a general PC cluster. We used HA-PACS (CPU: Intel Xeon E5-2670 2.60 GHz [eight CPU cores x



(a) RandomAccess (b) High-performance Linpack (c) Fast Fourier Transform

Figure 15: Performance Results on HA-PACS

two sockets/node], memory: DDR3 SDRAM 128 GB 51.4 GB/s/socket, network: Fat-Tree Infiniband QDR 4 GB/s x 2 (bi-directional) x two rails)[4] located at the University of Tsukuba. We used a maximum of 64 compute nodes (1,024 CPU cores) in this evaluation.

4.4.1 RandomAccess

We performed the RandomAccess on each of the CPU cores (flat-MPI). Figure 15(a) shows that the performance of XMP RandomAccess is slightly worse than that of modified hpcc-1.4.2. The reason for the low performance is the use of GASNet[10] as a one-sided communication layer for a coarray on a general PC cluster. Thus, the difference in performance between GASNet and MPI is less than that between the extended RDMA and MPI on the K computer.

4.4.2 High-performance Linpack

We performed the HPL on each socket (eight threads/process, two processes/node). Figure 15(b) shows the performances of HPL are approximately the same. Using 1,024 CPU cores (128 processes, 64 compute nodes), the performance of XMP implementation is 81% of the theoretical peak performance of the system. The reason for the good performance is that a compute node of HA-PACS has more memory than that of the K computer. Thus, the ratio of computation on HA-PACS is larger than that on the K computer, and the difference between the performances is slight.

4.4.3 Fast Fourier Transform

We performed FFT on each of the CPU cores. The reason for using flat-MPI was that the performance of flat-MPI was better than that of hybrid-parallelization. Figure 15(c) shows that both the performances of the FFT are almost the same. The reason for this is that thread-parallelization for the packing array in the hpcc-1.4.2 FFT was not performed in this evaluation (refer to Section 5.1.3).

5 Consideration

5.1 Productivity and Performance of XcalableMP

In this section, we discuss the productivity and performance of XMP for the implementations of the HPCC benchmarks in Section 4.

5.1.1 RandomAccess

In general, a coarray is a more convenient means for expressing communications in parallel programs than traditional MPI library routines. Therefore, it can be said that XMP provides good productivity in implementing the RandomAccess benchmark.

On the other hand, the XMP version with coarrays does not outperform the MPI, even though the extended RDMA interface used to implement coarrays in XMP could make the most use of the underlying network hardware and be faster than the vanilla MPI. There are two reasons for this. First, there is no way to complete memory operations from/to a certain image (or node in XMP) in the Fortran standard, and the **sync memory** statement being used, which completes all memory operations on the image, is too strong for such a purpose. Second, the current implementations of the **post** and **wait** directives for point-to-point synchronization are not optimal and could be implemented more efficiently with the extended RDMA interface.

5.1.2 High-performance Linpack

The **gmove** directive is useful for increasing the productivity. The panel broadcast operation is performed while maintaining the global image in Figures 7 and 8. In other words, the **gmove** directive automatically performs packing/unpacking data and communication for global array. However, it is difficult for programmers to understand in detail what operations and communications occur.

In order to increase the parallel efficiency of the HPL, it is known that scheduling of communication for panel broadcast is significant important. For example, a panel broadcast algorithm provided by the hpcc-1.4.2 HPL first sends the panel to a right-hand neighbor process, because the process can start a calculation using the received panel as soon as possible. If XMP uses this algorithm, XMP will use the function of the XMP coarray. In this case, the performance of the HPL will increase, but the complexity of the code will also increase.

5.1.3 Fast Fourier Transform

The SLOC of the subroutine "PZFFT1D0" written in XMP/Fortran is 65. On the other hand, the SLOC of an original "PZFFT1D0" written in Fortran with MPI is 101. Moreover, the code of the "PZFFT1D0" becomes simple because this code maintains a global image using the **gmove** and **loop** directives in Figure 11.

However, the main reason for the low performance is the **gmove** directive. Before the all-to-all communication, the transported data (two-dimensional array *A()* in Figure 12) must be packed in a one-dimensional array. This operation is performed internally in the **gmove** directive. In the hpcc-1.4.2 FFT, this operation is also performed in the same manner as the XMP version FFT. However, the hpcc-1.4.2 FFT performs this operation using thread-parallelization. The **gmove** directive has not yet supported internal thread-parallelization.

5.2 Comparison with Other PGAS Implementations

In this section, we present the performances and productivities of the HPCC benchmarks using other PGAS languages. Table 1 shows the performances and the SLOC of RandomAccess, HPL, and FFT at the HPCC Awards class 2 in 2011 and 2012, and the XMP implementations. Note that not only the SLOC but also the performance is only one of indicators of performance. This is because the performance of each benchmark is strongly related to the characteristics of the machine.

Table 1: Performance and SLOC of HPCC Awards Class 2 and XcalableMP in 2011 and 2012

Language	Machine	RandomAccess (GUPS) [SLOC]	HPL (TFlops) [SLOC]	FFT (TFlops) [SLOC]
X10	IBM Power 775	844 [143]	589 [708]	35 [236]
Chapel	Cray XE6	4 [112]	8 [658]	(NO DATA)
Charm++	BlueGene/P & Cray XT5	43 [138]	55 [1,770]	3 [185]
Coarray Fortran	Cray XT4 & XT5	2 [409]	18 [786]	0.3 [450]
XcalableMP	The K computer	163 [258]	543 [288]	24 [1,373]

Table 1 shows that the SLOC of HPL written in XMP are relatively good. The reason is that the **gmove** directive of XMP automatically generates complicated communication and pack/unpack operations. The SLOC of FFT written in XMP is a large number because we have implemented only the main kernel of all FFT subroutines. The other kernels are almost the same as the hpcc-1.4.2 FFT written in MPI. The performance of RandomAccess written in XMP is much worse than that written in X10. The reason is that IBM Power 775 has a significant high-speed interconnect (its peak bi-directional interconnect bandwidth of 192 GB/s)[27].

6 Conclusion and Future Research

The present paper describes the proposed implementations of RandomAccess, HPL, and FFT of the HPCC benchmarks using XMP. Through these implementations, we demonstrate that the XMP programming model has good productivity for an HPC system. Moreover, we evaluated the performances on the K computer, which is a leadership HPC system. The results using a maximum of 262,144 CPU cores show that XMP has scalable performances. There are two important reasons for the good performance. First, XMP coarray syntax directly uses a high-speed extended RDMA interface on the K computer. Second, XMP can tune the application while being aware of local memory because XMP provides functions to obtain local memory information for a global array. These results show that XMP has good strategies for high performance and productivity of HPC applications.

In the future, several investigations have been described as follows. First, we will rewrite part of the code on the local-view model in order to increase the performance. Second, the internal packing operation in the **gmove** directive will support thread parallelization. Third, we will develop real applications using XMP. Finally, we will implement applications using XMP and OpenACC on an accelerator cluster and evaluate their productivity and performance.

Acknowledgments

- Modification of the two functions for sorting and updating of the local table in RandomAccess were performed in cooperation with Fujitsu Limited.

- The authors would like to thank Ikuo Miyoshi who belongs to Fujitsu Limited for his lecture on HPL.

- The present study was supported by the “Feasibility Study on Future HPC Infrastructure” project funded by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[1] `http://www.netlib.org/blas/`.

[2] `http://www.netlib.org/scalapack/`.

[3] Ffte: A fast fourier transform package. `http://www.ffte.jp`.

[4] Ha-pacs. `http://www.ccs.tsukuba.ac.jp/CCS/eng/research-activities/projects/ha-pacs`.

[5] Hpc challenge benchmarks. http://icl.cs.utk.edu/hpcc/.

[6] Omni xcalablemp compiler. http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/xcalablemp/.

[7] Openacc home. http://www.openacc-standard.org.

[8] Pc cluster consortium. http://www.pccluster.org/en/.

[9] Top500 supercomputer sites. http://www.top500.org.

[10] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In The 20th Int'l Parallel and Distributed Processing Symposium (IPDPS), 2006.

[11] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[12] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.

[13] UPC Consortium. Upc language specifications. Technical Report LBNL-59208, Berkeley National Laboratory, 2005. http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf.

[14] Frontiers of Massively Parallel Computation. *Communication overhead on CM5: an Experimental Performance Evaluation*, 1992.

[15] FUJITSU LIMITED. *Parallelnavi Technical Computing Language MPI User's Guide*, 2013.

[16] Jorge Gonzlez-Domnguez, Mara J. Martn, Guillermo L. Taboada, Juan Tourio, Ramn Doallo, Damin A. Malln, and Brian Wibecan. Upcblas: a library for parallel matrix computations in unified parallel c. *Concurr. Comput. : Pract. Exper.*, 24(14):1645–1667, September 2012.

[17] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[18] Charles H. Koelbel, David B. Loverman, Robert S. Shreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[19] Jinpil Lee. *A Study on Productive and Reliable Programming Environment for Distributed Memory System*. PhD thesis, University of Tsukuba, 2012.

[20] Jinpil Lee, Minh Tuan Tran, Tetsuya Odajima, Taisuke Boku, and Mitsuhisa Sato. An extension of xcalablemp pgas lanaguage for multi-node gpu clusters. In *Proceedings of the 2011 international conference on Parallel Processing*, Euro-Par'11, pages 429–439, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] Nakao Masahiro, Lee Jinpil, Boku Taisuke, and Sato Mitsuhisa. Xcalablemp implementation and performance of nas parallel benchmarks. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 11:1–11:10, New York, NY, USA, 2010. ACM.

[22] Nakao Masahiro, Lee Jinpil, Boku Taisuke, and Sato Mitsuhisa. Productivity and performance of global-view programming with xcalablemp pgas language. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 402–409, Washington, DC, USA, 2012. IEEE Computer Society.

[23] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A non-uniform-memory-access programming model for high-performance computers. *THE JOURNAL OF SU-*

PERCOMPUTING, 10:169–189, 1996.

[24] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[25] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification, 2013. http://x10.sourceforge.net/documentation/languagespec/x10-231.pdf.

[26] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[27] Olivier Tardieu, David Grove, Bard Bloom, David Cunningham, Benjamin Herta, Prabhanjan Kambadur, Vijay A. Saraswat, Avraham Shinnar, Mikio Takeuchi, and Mandana Vaziri. X10 for productivity and performance at scale. RC25334 (WAT1210-081), October 2012.

[28] XcalableMP Specification Working Group. Xcalablemp specification version 1.1, 11 2012. `http://www.xcalablemp.org/spec/xmp-spec-1.1.pdf`.

[29] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.

PGAS implementation of SpMVM and LBM using GPI

Faisal Shahzad, Markus Wittmann, Moritz Kreutzer, Thomas Zeiser,
Georg Hager and Gerhard Wellein

Erlangen Regional Computing Center
University of Erlangen-Nuremberg, 91058 Erlangen, Germany
{faisal.shahzad, markus.wittmann, moritz.kreutzer, thomas.zeiser,
georg.hager, gerhard.wellein}@fau.de

Abstract

GPI is a PGAS model based library that targets to provide low-latency and highly efficient communication routines for large scale systems. We compare and analyse the performance of two algorithms, which are implemented with GPI and MPI. These algorithms are a sparse matrix-vector-multiplication (SpMVM) and a fluid flow solver based on a lattice Boltzmann method (LBM). Both algorithms are purely memory-bound on a single node, whereas at the large scale, the communication between the processes becomes more significant. GPI, in principle, is fully capable of performing communication alongside computation. Both the algorithms are modified to leverage this feature. In addition to the naïve approach with blocking calls in MPI, the algorithms are also evaluated using non-blocking calls and explicit asynchronous progress via an external library. We conclude that GPI implementations handle non-blocking asynchronous communication very effectively and thus hiding communication costs.

1 Introduction and background

In order to increase the efficiency and scalability of MPI-like programming environments, many developments have been made over the past several years. Using Partitioned Global Address Space (PGAS)[10] languages is one part of these efforts. The PGAS model provides a shared memory view for distributed memory systems, thus making it possible to access (read/write) memory of the remote processes without their active involvement. The main motivation behind the PGAS programming model is to increase the simplicity of codes and to provide better efficiency and scalability at the same time. Some of the PGAS languages gained some popularity like Coarray Fortran[3], Unified Parallel C[17], and Chapel[2]. Some other implementations provide a PGAS communication model via API calls. This avoids the need to learn a new language and a complete rewrite of applications. The Global address space Programming Interface (GPI)[4] and Global Arrays Toolkit[5] are examples of such PGAS based APIs.

The GPI library is a relatively new addition to the PGAS programming model libraries. Developed by Fraunhofer ITWM[1], it focuses on providing low-latency, high speed communication routines for large scale systems.

In the scope of this paper, we implement two algorithms using GPI. The first is a sparse matrix-vector-multiplication (SpMVM) algorithm. SpMVM is a significant operation which occurs in many scientific algorithms quite often and makes up a large amount of the overall application runtime. The second algorithm is a fluid flow solver based on a lattice Boltzmann method (LBM)[14]. Due to its effective algorithmic implementation and ease of parallelization,

[1] Fraunhofer Institute for Industrial Mathematics (ITWM): http://www.itwm.fraunhofer.de/en

LBM has gained significant importance in scientific and research communities. We compare the performance and scalability of our GPI vs. MPI implementations and extend our previous work [12] where MPI and Coarray Fortran were compared.

Related Work: Grünewald [6] used a stencil based application (BQCD) to compare GPI and MPI implementations. A performance advantage of 20-30 % is achieved for the GPI code through its ability to perform communication asynchronously to the work the application performs. In [13] Simmendinger et al. implemented the unstructured CFD solver TAU using GPI and receives a far better speedup as compared to MPI and hybrid MPI/OpenMP implementations. A similar comparison for a parallel adaptive search algorithm is presented by Machado et al. in [8], who achieve a better speedup than the MPI implementation for some test cases. The core reason for the improvement in application performance with GPI is the reformulation of algorithms to benefit from the one-sided asynchronous communication capabilities of GPI.

An important point in optimizing communication is to create or leverage communication hiding. Hereby the communication library (e.g. PGAS or MPI) communicates in the background while the application continues with its own work. In this case asynchronous progress of the message transfer is performed and communication and computation overlap. The MPI standard does not require this behavior (also not for the non-blocking point-to-point semantic) and only high-quality implementations provide support for this. A rough evaluation of MPI implementations concerning this feature can be found in [7] and [20]. To provide asynchronous progress, even if it is not supported by the MPI implementation, the APSM library (Asynchronous Progress Support for MPI) [20] can be used, which requires a thread-safe MPI library. Transparently to the user, non-blocking MPI communication calls are intercepted by the library and an own progress thread is used to drive the message transfer in the background. The pinning (a.k.a. affinity) of the progress thread can be set by an environment variable. We test the GPI performance against the MPI versions of the algorithms with and without APSM-library support.

The main contributions of this paper are:

1. Implementation of SpMVM with GPI.
2. Implementation of LBM with GPI.
3. Optimization of GPI based SpMVM and LBM implementations to achieve asynchronous communication.
4. A thorough performance comparison of GPI with their respective MPI implementations.
5. The performance benchmark of MPI with APSM support and its comparison with GPI performance.

This paper is structured as follows. In Sect. 2, we present an introduction to GPI. The experimental framework is presented in Sect. 3. In Sect. 4 the GPI implementation of SpMVM is elaborated. Further the performance benchmarks are conducted and compared with the MPI implementation. The same is done for the LBM code in Sect. 5. Finally, Sect. 6 gives the summary and concludes the paper.

2 Global address space Programming Interface (GPI)

The Global address space Programming Interface (GPI) is a library based on the PGAS style programming model for C/C++ and Fortran. The GPI API consists of a set of basic routines. Its communication layer can take full advantage of the hardware capabilities to utilize remote

direct memory access (RDMA) for spending no CPU cycles on communication. Each GPI process has a *global* and *local memory*. The global memory can be accessed by other processes, i.e. the data that has to be shared with other processes must be located in this memory region. The local memory is private to each process and hence is not accessible to other processes. The amount of global memory for each process must be specified at the start of the application when GPI is initialized by the *startGPI* call (similar to *MPI_Init*).

The API of GPI provides one-sided and the usual two-sided communication calls. The one-sided calls consist of *readDmaGPI* and *writeDmaGPI*. Both of these calls are non-blocking and require a wait call (*waitDmaGPI*) to ensure the accomplishment of the read/write operation. The two-sided communication calls are *sendDmaGPI* and *recvDmaGPI*. All communication calls can only operate on the global memory. The only exception to this is *allReduceGPI* call which can use local memory as well.

GPI only supports one GPI process per socket (or node), thus for multi-core environments, GPI relies on a threading module called Multi-core Threading Package (MCTP) for intra-socket resource utilization. The MCTP threading module also is developed by Fraunhofer ITWM and provides fine NUMA-aware control over threads. The more widely used OpenMP programming model can also be used as an alternative threading package. A minimum of two nodes are required to run a GPI program. GPI supports InfiniBand and Ethernet (RDMAoE, RoCE) interconnects.

The present version (1.0) of GPI also provides basic fault-tolerance on a process level. Unlike an MPI application, if one or more processes of a GPI application fail, the remaining processes can proceed with their work as usual.

3 Experimental Framework

For performance evaluation, we have used RRZE's LiMa[2] cluster. This cluster comprises of 500 compute nodes equipped with two Intel Xeon 5650 "Westmere" CPUs (six physical cores, two-way SMT cores) running at the base frequency of 2.66 GHz with 12 MB shared cache per chip. The ccNUMA system has two locality domains, each with 12 GB RAM (24 GB in total). The STREAM (scale) benchmark [9] achieves a bandwidth of around 40 GB/s (20 GB/s per socket). Simultaneous multithreading (SMT) and "Turbo Mode" are enabled. The system is equipped with Mellanox QDR InfiniBand (IB) and GBit Ethernet interconnects. GPI can only use the IB interconnect as the onboard GBit chips do not support RDMAoE or RoCE.

In the scope of this paper, we present the implementation details and results of the following two algorithms.

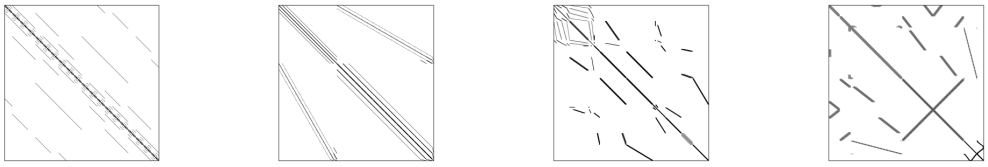
SpMVM: The Sparse Matrix-Vector-Multiplication is a significant operation which occurs in many scientific applications. Mostly it covers a significant portion of the overall computation time. Thus its efficiency is critical to the application performance. A SpMVM operation consists of $\vec{y} = \mathbf{A}\vec{x}$, where $\mathbf{A}$ is an $n \times n$ dimensional matrix, and $\vec{x}, \vec{y}$ are n dimensional vectors. The operation can be written as

$$y_i = \sum_j (A)_{i,j} \cdot x_j. \tag{1}$$

[2]LiMa cluster at the Erlangen Regional Computing Center (RRZE): `http://www.hpc.rrze.fau.de/systeme/lima-cluster.shtml`

Matrix	Dimension	Avg. NNZ per row	size in MB
RRZE3	$6.2 \cdot 10^6$	19	1530
DLR1	$2.8 \cdot 10^5$	144	642
HV15R	$2 \cdot 10^6$	140	4545
RM07R	$3.8 \cdot 10^5$	98	602

Table 1: The matrices used for benchmarks.



(a) Matrix RRZE3 (b) Matrix DLR1 (c) Matrix HV15R (d) Matrix RM07R

Figure 1: The sparsity structure of the benchmarked matrices.

In case of a dense matrix, it is obvious to store the matrix in an $n \times n$ array. On the other hand, for sparse matrices, the effective storage strategy of the matrix is critical for performance. Some of the most famous matrix storage schemes are Compressed Row Storage (CRS), Compressed Columns Storage (CCS), Ellpack-Itpack, Jagged-Diagonal, Blocked Compressed Row Storage (BCRS), Compressed Diagonal Storage (CDS), etc [15]. For CPUs, the CRS storage scheme is known for its best all-round performance for non-structured sparse matrices. Therefore, in this paper, we follow this format. This scheme consists of three one dimensional arrays: *values*, *column_index*, and *row_pointer*. The *values* array consists only of non-zero values of the matrix. The *column_index* array stores the column index of the corresponding non-zero matrix value. The *row_pointer* determines the number of non-zeros in a row. The size of *values* and *column_index* arrays is equal to the number of non-zeros in the matrix. The size of *row_pointer* array is equal to $n + 1$.

The parallel MPI and GPI implementation details and results are provided in Sec. 4. We have performed the benchmarks using four different sparse matrices: RRZE3, DLR1, HV15R[16], and RM07R[16]. Table 1 shows the details of the matrices used whereas sparsity structure of the matrices is shown in Fig. 1.

LBM: The lattice Boltzmann Method[14] is used as a fluid flow solver. In the recent past it has gained significant popularity in science and research communities also due to the fact that it is simple to parallelize. LBM can be seen as a Jacobi-like stencil algorithm, but with two major differences: (1) each cell has not only one, but (as in our case) 19 values and (2) no values read are reused during the same iteration over the lattice. Our implementation uses the D3Q19 model[11] with the BGK collision operator[1]. In each iteration, the data is read from the 4-D source lattice (three spatial dimensions plus one for the 19 cell values) and modified values are written to the 4-D destination grid. The update of one cell is performed by reading one value of each of the cell's 19 surrounding neighbors. Out of these values new ones are computed, which are used to update the cell's own values in the destination lattice. Thereby the values are arranged in a structure-of-array data layout (for details see [18]).

In this paper, we have used a prototype application based on LBM which simulates a 3-D

```
SpMVM_sync(mat,rhs,res)
{
        rhs->communicate_remote_RHS_blocking();
        spMVM(mat, rhs, res);
}
```

Listing 1: The SpMVM function with synchronous MPI communication.

lid-driven cavity problem. The implementation details of its MPI and GPI parallelism are given in Sec.5.

For both the applications, the performance of the five following cases is examined:

1. Blocking MPI communication (naïve MPI case)
2. Non-blocking MPI communication
3. Non-blocking MPI communication with APSM library
4. Synchronous GPI communication
5. Asynchronous GPI communication

Our evaluation of the benchmarks is based on absolute performance of the application and the degree of overlap efficiency the library can offer. In addition, the scaling behavior of the overlap efficiency is also examined. For benchmarking, the Intel C++ compiler (version 12.1.11) with Intel MPI (version 4.0.3.008) was used. The pinning of the threads is explicitly performed from within the code by *sched_setaffinity()*.

4 SpMVM

The parallel implementation of the SpMVM algorithm is based upon distributing the matrix rows amongst the processes. Each process then calculates its result vector $\vec{y}$ corresponding to its assigned rows. The division of the number of rows per process can be performed in two ways. A naïve way is to divide the number of rows evenly between all processes. For sparse matrices, this approach can potentially result in work imbalance amongst processes. For optimal workload distribution, our implementation is based upon having (more or less) equal number of non-zeros for each process. Each process owns the part of the right hand side (RHS) $\vec{x}$ that corresponds to its rows of the matrix. The calculation of the result vector $\vec{y}$ can be seen as a combination of two parts i.e. a "local part" and a "non-local part". The part for which the RHS values are local to the process is regarded as local part. The other part, for which the RHS values need to be fetched from other processes is regarded as non- local part.

In a simple synchronous communication approach, these parts are combined. Thus, communication of RHS is required before performing the sparse-matrix-vector-multiplication. Listings 1 and 2 show the SpMVM function in a synchronous communication setting for MPI and GPI, respectively. In the MPI case, the communication is two-sided and no global synchronization is required. On the contrary, two synchronization calls are required for the GPI case. The first synchronization ensures the completion of communication on all processes. This guarantees that each process has updated RHS values received by remote processes. The second synchronization is required before the next SpMVM. After that all processes have finished the previous SpMVM iteration and have up-to-dated RHS values.

In an asynchronous (non-blocking) communication setting, the local and non-local parts are computed separately. The communication request for reading the non-local RHS is placed before

```
SpMVM_sync(mat,rhs,res)
{
        rhs->communicate_remote_RHS_one_sided();
        rhs->wait();
        sync();
        spMVM(mat, rhs, res);
        sync();
}
```

Listing 2: The SpMVM function with synchronous GPI communication.

```
SpMVM_async(mat,rhs,res)
{
        rhs->communicate_remote_RHS_non-blocking();
        spMVM_local(mat, rhs, res);
        rhs->wait();
        spMVM_remote(mat, rhs, res);
}
```

Listing 3: The SpMVM function with asynchronous MPI communication.

the local-SpMVM kernel. A wait routine ensures the completion of this communication request. After receiving the non-local RHS values, the non-local part can now be computed. Listings 3 and 4 show the pseudo code of the SpMVM function with asynchronous communication for MPI and GPI, respectively. As in the previous case, no global synchronization is required for the MPI case. GPI on the other hand, requires the first synchronization call before computing non-local part. The second synchronization is needed before the next iteration.

The GPI asynchronous communication has two effects on the execution of the code. First, the overlapping of communication and computation, which results in better performance. The second effect is due to the introduction of a global synchronization call between local and non-local SpMVM parts. This is matrix dependent and can be large if the ratio of local to non-local NNZ amongst all processes is not even. If this imbalance is big, it can result in large overhead.

For benchmarking, we have followed a hybrid implementation i.e. MPI/OpenMP and GPI/OpenMP. The parallelization within the socket is performed using OpenMP threads. Figure 2 shows the performance comparison of five different cases (as described in Sec. 3) for different matrices. This benchmark has been performed on 32 LiMa nodes with 64 processes each having 6 threads. The performance of non-blocking MPI is nearly similar to naïve MPI (blocking MPI) for all matrices. This shows that Intel MPI (version 4.0.3) can not perform

```
SpMVM_async(mat,rhs,res)
{
        rhs->communicate_remote_RHS_one_sided();
        spMVM_local(mat, rhs, res);
        rhs->wait();
        sync();
        spMVM_remote(mat, rhs, res);
        sync();
}
```

Listing 4: The SpMVM function with one-sided asynchronous GPI communication.

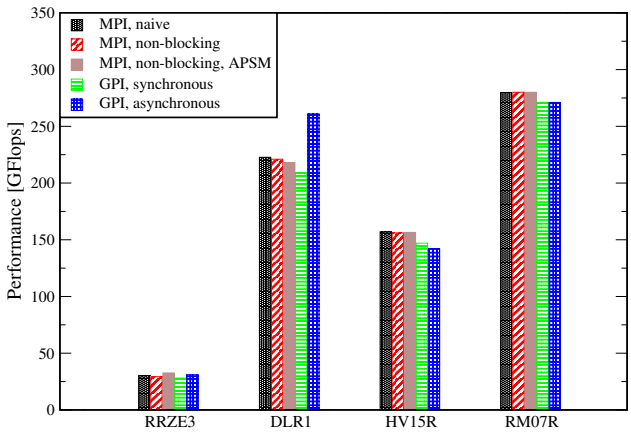


Figure 2: The performance comparison of SpMVM with naïve MPI, non-blocking MPI , non-blocking MPI with APSM, synchronous GPI, and asynchronous GPI cases for four different matrices on 32 LiMa nodes.

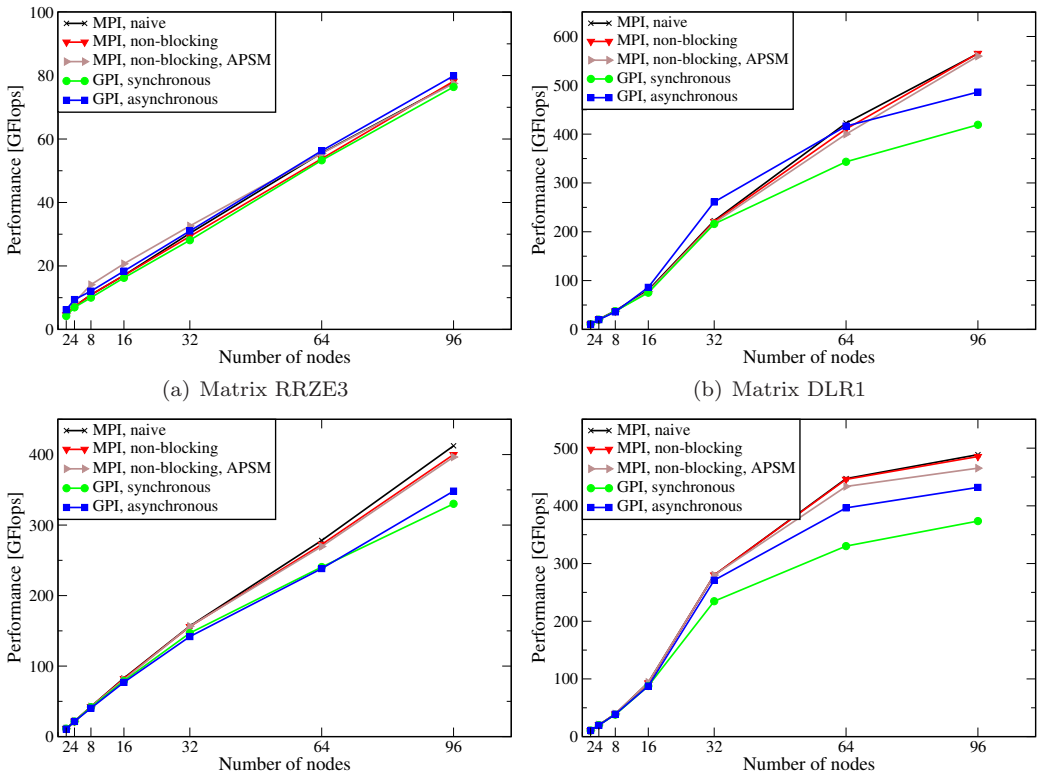
the non-blocking communication calls in an asynchronous way. The usage of APSM library boosts the performance only in the case of RRZE3. For RRZE3 and DLR1 matrices, the GPI asynchronous communication performs better than the naïve MPI case. The APSM support case with RRZE3 matrix is even better than the asynchronous GPI. The synchronization cost in case of GPI introduces large overhead in the case of HV15R and RM07R and hides the advantage gained due to asynchronous communication. Thus, the performance for HV15R and RM07R is lower than the MPI cases.

Figure 3 shows the performance comparison between various cases of MPI vs. GPI in case of strong scaling from 2 to 96 LiMa nodes. As per GPI requirement, the baseline for this strong scaling starts with two nodes. For all matrices, the baseline performance of MPI and GPI is nearly the same except for RRZE3 where asynchronous GPI performs 35 % better than the naïve MPI case. Around 32 nodes, the matrices DLR1 and RM07R fit into the L3 cache completely and thus the strong scaling performance does not increase in a linear fashion from this point on. The asynchronous GPI performance stays similar to the MPI performance for smaller number of nodes. For more than 32 nodes, the global synchronization in each iteration for GPI becomes more expensive and starts to reduce the performance.

5 Lattice Boltzmann Method

The domain in our test application consists of a 3-D lid-driven cavity. A hybrid parallel model is followed using a MPI/OpenMP and GPI/OpenMP. For parallelization, the domain is divided into slices in the Z-direction. Thus the ghost elements are exchanged in the Z-direction. The size of this communication depends on the number of cells in X and Y-directions. The runtime of the benchmark depends on the time-steps and the number of domain cells.

In a typical synchronous stream-collide based implementation, first the domain cells are updated in a stream-collide step. Boundary layers are then exchanged to update the ghost cells of each process before moving on to the next iteration. Listing 5 shows the algorithm for an LBM iteration loop with synchronous communication. As we have followed one-sided communication routines (read/write) in GPI, a barrier is essential before performing communication (i.e.



(a) Matrix RRZE3 (b) Matrix DLR1

(c) Matrix HV15R (d) Matrix RM07R

Figure 3: The MPI vs. GPI SpMVM performance comparison in case of strong scaling for different matrices.

```
for(int t=1; t <= timesteps; ++t)
{
        update_cells();
        barrier();
        exchange_ghost_cells();
}
```

Listing 5: LBM iteration loop with synchronous communication. The barrier is essential before one-sided communication is performed (i.e. GPI case).

updating ghost cells) in order to make sure that the target buffer has already been used and is ready for the next update. As communication can only take place between global memories of processes, all the distribution-functions (including ghost elements) are allocated in the global memory of GPI. This avoids the in-memory copy of the ghost-elements from local to the global memory.

In order to take the benefit of GPI's capability to perform communication asynchronously in the background of the application, we adapted the algorithm to make this overlap possible. This involved splitting the stream-collide routine into two parts. The first part includes performing stream-collide on the boundary (next-to-ghost) cells. After performing this step, the

```
for(int t=1; t <= timesteps; ++t)
{
        update_boundary_cells();
        exchange_ghost_cells_begin();
        update_inner_cells();
        exchange_ghost_cells_end();
}
```

Listing 6: Pseudo code of LBM iteration loop with asynchronous communication (computation-communication overlap)

communication of ghost-layers is initiated. In the second stream-collide stage, the computation on the inner cells of the domain is performed while the ghost-layers communication takes place in the background. The wait call after the second stream-collide step makes sure that communication has been completed. Listing 6 shows a simple pseudo code for such a case.

For GPI, as the communication is one-sided (read/write), the synchronization between the processes is essential to make sure that the read values from the neighboring process are valid values (i.e. the remote process for the read operation has already updated the values to be read). One way to make this synchronization possible is by introducing a global barrier. In order to avoid the cost of global synchronization, we have implemented a synchronization scheme only between the communicating neighbor processes. The approach is similar to the one presented in [6]. It has been combined with a relaxed synchronization approach [19] to further enhance the communication performance. In a relaxed synchronization approach, each process copies its owed values in a separate so called transfer buffer, which is located in the global memory. One flag *boundary_ready* per process and direction is required in the global memory. The local process sets its flags after its updated data values are written into the transfer buffer. A remote process, who wants to fetch the local process boundary values, polls on this flag. When it becomes ready the remote process reads the desired values and unsets the flag again. The local process waits for the flag to be unset before it continues with the next iteration. Listing 7 shows the LBM iteration loop with such a relaxed synchronization. All cell data is stored in the local memory and only the transfer buffers and synchronization flags are allocated in the global memory.

For the LBM case in particular, some overhead gets inherently induced with the adjustment of the algorithm for GPI in order to achieve the communication overlap. The first overhead originates from the in-memory coping of the transfer buffers (i.e. from local to global memory). The second, rather small overhead is introduced by splitting the "stream-collide" routine which updates the cells and thus causing less efficient cache usage.

Figure 4 presents the results of weak scaling up to 96 LiMa nodes for MPI (naive, non-blocking and non-blocking with APSM) and GPI (synchronous and asynchronous). The domain size of $2100 \times 2100 \times 24$ cells is selected for the base case of 2 nodes. As the communication is carried out over an InfiniBand network, a large cross-section of the domain is chosen (2100×2100) to have a significant contribution of communication to the overall runtime. This way, the communication hiding can be clearly seen. For weak scaling the domain size is scaled in Z-direction. The baseline case for this performance benchmark is the naïve MPI case (i.e. no overlap between computation and communication). For the case of MPI with non-blocking communication, the performance remains largely unchanged. The case with APSM support (MPI, non-blocking, APSM) shows a performance improvement. Naïve GPI beats the naïve MPI implementation as the number of nodes get larger. The best performance is achieved with

```
for(int t=1; t <= timesteps; ++t)
{
        update_boundary_cells();
        copy_boundary_cells_to_comm_buffer();
        boundary_ready[local_rank][EAST] = 1;
        boundary_ready[local_rank][WEST] = 1;

        wait_for(boundary_ready[remote_rank_east][WEST] == 1);
        wait_for(boundary_ready[remote_rank_west][EAST] == 1);

        read_remote_boundary_cells();

        boundary_ready[remote_rank_east][WEST]=0;
        boundary_ready[remote_rank_west][EAST]=0;

        update_inner_cells();

        wait_for(boundary_ready[local_rank][EAST] == 0);
        wait_for(boundary_ready[local_rank][WEST] == 0);
}
```

Listing 7: Pseudo code of LBM iteration loop having asynchronous communication and relaxed synchronization.

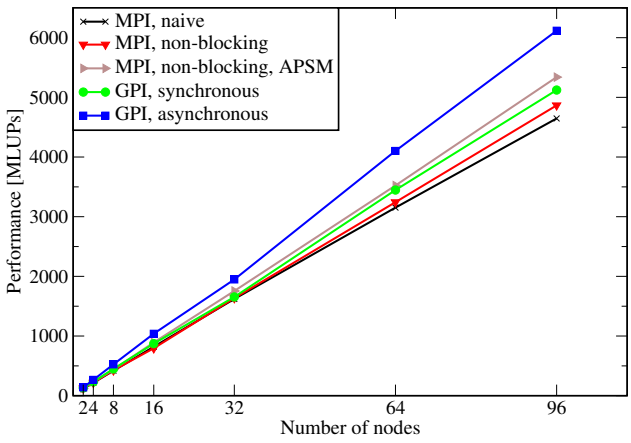


Figure 4: Performance of LBM implemented with MPI and GPI on LiMa in the case of weak scaling with $2100 \times 2100 \times 12$ cells per process.

the algorithm adapted for asynchronous communication with GPI. The performance difference to all other variants becomes increases for larger number of nodes. With 96 nodes, the LBM implementation with asynchronous GPI case is $\approx 30\,\%$ better than the naïve MPI one.

The communication time can only be fully overlapped as long as it is smaller than the computation time. In order to check the efficiency of overlap, we define the *overlap fraction* μ as follows:

$$\mu = \frac{T_{sync} - T_{async}}{T_{comm}} \tag{2}$$

Here, T_{sync} and T_{async} represent the total runtime synchronous and asynchronous communica-

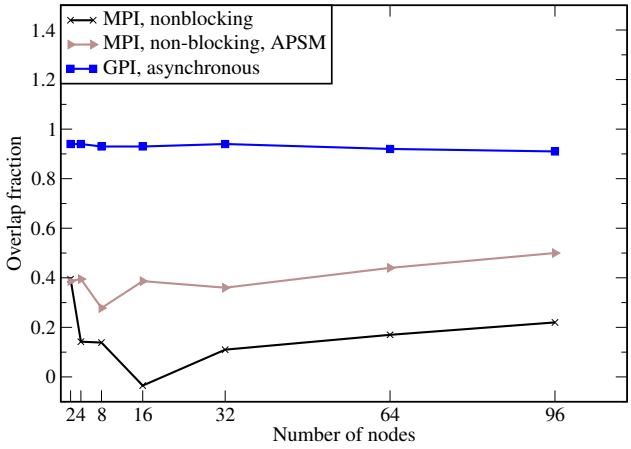


Figure 5: Fraction of communication that can be overlapped with computation for LBM implemented with MPI and GPI variants.

tion respectively. T_{comm} represents the pure communication time for the case without overlap. A higher number of μ indicates more efficient communication computation overlap. If computation gets slower by the overlapped communication, the T_{async} gets larger and effects the overlap fraction.

Figure 5 shows the communication overlap fraction μ for LBM benchmark up to 96 nodes. The overlap with non-blocking MPI is low and inconsistent. Utilizing the APSM library improves this behavior, but yet not to the best. The duration for communication completion get reduced, but simultaneously the computation time increases. Hence no optimal overlap fraction is reached. The best results are achieved with asynchronous GPI with an overlap fraction close to 95 %.

6 Summary

In order to improve the scalability, performance and ease of programmability for very large scale computing clusters, the usage of Partitioned Global Address Space (PGAS) is one implementation candidate. The GPI library is a relatively new addition to the PGAS programming model libraries. In this paper, we have implemented a sparse matrix-vector-multiplication (SpMVM) and a lattice Boltzmann method based application with GPI and MPI. For both algorithms a comparative performance study was concluded. The results revealed that in order to fully utilize the potential of GPI, algorithms have to be adapted to allow for a communication-computation overlap. In principle, this is also possible for MPI, but support for asynchronous communication depends on the used MPI implementation. With SpMVM the performance benefit with GPI is matrix dependent and the global synchronization reduces the performance on large number of nodes. In the case of LBM the expensive global synchronization could be replaced by a more relaxed synchronization scheme, which is the reason for the significant performance-gain over the non-blocking MPI implementation.

Acknowledgment

We are grateful to Fraunhofer-ITWM and Scapos for providing an evaluation license of GPI. We specially thank to Dr. Grünewald for his support and helpful discussions.

This work was supported by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA) and the Federal Ministry of Education and Research (BMBF) under project "A Fault Tolerant Environment for Peta-scale MPI-solvers" (FETOL) (grant No. 01IH11011C).

References

[1] P. L. Bhatnagar, E. P. Gross, and M. Krook. A model for collision processes in gases: Small amplitude processes in charged and neutral one-component systems. *Phys. Rev.*, 94:511–525, 1954.

[2] Chapel. `http://chapel.cray.com/`.

[3] Co-Array Fortran. `http://www.co-array.org/`.

[4] Fraunhofer IWTM: Global Address Space Programming Interface (GPI). `http://www.gpi-site.com/cms/?q=node/15`.

[5] Global Arrays Toolkit. `http://www.emsl.pnl.gov/docs/global/`.

[6] D. Grunewald. BQCD with GPI: A case study. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 388–394, 2012. doi: `10.1109/HPCSim.2012.6266942`.

[7] G. Hager, G. Schubert, T. Schoenemeyer, and G. Wellein. Prospects for Truly Asynchronous Communication with Pure MPI and Hybrid MPI/OpenMP on Current Supercomputing Platforms. In *Cray Users Group Conference 2011*, Fairbanks, AK, USA, 2011.

[8] Rui Machado, Salvador Abreu, and Daniel Diaz. Parallel Local Search: Experiments with a PGAS-based programming model. *CoRR*, 2013.

[9] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007.

[10] Partitioned Global Address Space. `http://www.pgas.org/`.

[11] Y. H. Qian, D. D'Humires, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)*, 17(6):479–484, 1992.

[12] Klaus Sembritzki, Georg Hager, Bettina Krammer, Jan Treibig, and Gerhard Wellein. Evaluation of the Coarray Fortran Programming Model on the Example of a Lattice Boltzmann Code. In *PGAS12*, 2012.

[13] Christian Simmendinger, Jens Jgerskpper, Rui Machado, and Carsten Lojewski. A PGAS-based Implementation for the Unstructured CFD Solver TAU. In *PGAS11*, Galveston Island, Texas, USA, 2011.

[14] S. Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Oxford University Press, 2001.

[15] Faroogh Tavakoli. Parallel sparse matrix-vector multiplication. Master's thesis, Royal Institute of Technology, Stockholm, Sweden, 1997.

[16] The University of Florida Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices/`.

[17] Unified Parallel C (UPC). `http://upc.gwu.edu/`.

[18] G. Wellein, T. Zeiser, G. Hager, and S. Donath. On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids*, 35(8–9):910–919, 2006.

[19] Markus Wittmann, Georg Hager, Jan Treibig, and Gerhard Wellein. Leveraging Shared Caches for Parallel Temporal Blocking of Stencil Codes on Multicore Processors and Clusters. *Parallel Processing Letters*, 20(4):359–376, 2010. doi: 10.1142/S0129626410000296.

[20] Markus Wittmann, Georg Hager, Thomas Zeiser, and Gerhard Wellein. Asynchronous MPI for the Masses. 2013. arXiv:1304.7664.

Optimizing Collective Communication in OpenSHMEM *

Jithin Jose, Krishna Kandalla, Jie Zhang, Sreeram Potluri and Dhabaleswar K. (DK) Panda

Department of Computer Science and Engineering, The Ohio State University
{jose, kandalla, zhanjie, potluri, panda}@cse.ohio-state.edu

Abstract

Message Passing Interface (MPI) has been the de-facto programming model for scientific parallel applications. However, data driven applications with irregular communication patterns are harder to implement using MPI. The Partitioned Global Address Space (PGAS) programming models present an alternative approach to improve programmability. OpenSHMEM is a library-based implementation of the PGAS model and it aims to standardize the SHMEM model to achieve performance, programmability, and portability. However, since OpenSHMEM is an emerging standard, it is unlikely that entire applications will be re-written with it. Instead, unified communication runtimes have paved the way for a new class of hybrid applications that can leverage the benefits of both MPI and PGAS models. Such unified runtimes need to be designed in a high performance, scalable manner to improve the performance of emerging hybrid applications. Collective communication primitives offer a flexible, portable way to implement group communication operations and are supported in both MPI and PGAS programming models. Owing to their advantages, they are also widely used across various scientific parallel applications. Over the years, MPI libraries have relied upon aggressive software-/hardware-based and kernel-assisted optimizations to deliver low communication latency for various collective operations. However, there is much room for improvement for collective operations in state-of-the-art, open-source implementations of OpenSHMEM. In this paper, we address the challenges associated with improving the performance of collective primitives in OpenSHMEM. Further, we also explore design alternatives to enable collective primitives in OpenSHMEM to directly leverage the designs available in the MVAPICH2 MPI library. Our experimental evaluations show that our designs significantly improve the performance of the OpenSHMEM's broadcast and collect operations. Our designs also improve the performance of the Graph500 benchmark using by up to 57%, with 4,096 OpenSHMEM processes.

1 Introduction and Motivation

The Message Passing Interface (MPI) [20] has been a popular programming model for High Performance Computing (HPC) applications for the last couple of decades. MPI has been very successful in implementing regular, iterative parallel algorithms with well-defined communication behaviors. Data-driven applications often pose challenges associated with load balancing and often exhibit irregular communication patterns. These issues are harder to address with a traditional message-passing programming paradigm. The Partitioned Global Address Space (PGAS) programming models present an alternative approach compared to message passing and are believed to improve programmability of such applications. However, the existing PGAS

*Thisresearch is supported in part by National Science Foundation grants #OCI-0926691, #OCI-1148371 and #CCF-1213084.

models are still emerging and being standardized, whereas the MPI models are much more widely adopted. The MPI models are also extensively tuned and can deliver superior performance and scalability. Owing to these reasons, it is unlikely that applications will be designed solely with the PGAS models in the future. It is more likely that applications will continue to be written with MPI as their primary programming model; but, parts of the applications will be adapted to use a suitable PGAS model, such as OpenSHMEM or UPC, leading to a class of "hybrid" applications. This trend has also paved the way for unified communication runtimes, such as, MVAPICH2-X [22], which allow applications to leverage the best of both MPI and PGAS models.

Collective communication primitives offer a flexible, portable way to implement group communication operations. Owing to their advantages, collective operations are supported across both MPI and PGAS models. They are also widely used across various scientific applications [9, 14]. Most MPI stacks implement collective communication using point-to-point operations. However, with the increasing use of multi-core platforms, high-performance MPI implementations have incorporated optimizations specific to multi-core architectures [19, 15, 6]. Some implementations of MPI, such as those on IBM Blue Gene, leverage specific network and system features to optimize latency of collective operations [8, 16]. MPI implementations also rely on kernel-assisted mechanisms to improve the performance of collective operations [17].

The MPI standard defines a high-level communicator construct to define the scope of communication operations and to facilitate specific optimizations for collective operations. MPI libraries typically construct sub-communicators and implement collective operations in a hierarchical, multi-core aware manner to deliver low communication overheads. However, the communicator object is fairly robust and affects the development of irregular applications, which may involve communication operations with varying process groups. Hence, PGAS models, such as OpenSHMEM, allow applications to specify the set of processes participating in a collective in a "loosely" defined manner, through `PE_start`, `PE_size`, and `logPE_stride` variables. An OpenSHMEM implementation can choose to construct a communication tree to implement a collective operation, such as, broadcast, based on these parameters. Or, implementations may choose to implement the collective in a simple linear manner through series of puts and gets. Invariably, such designs are not multi-core-aware, and do not use advanced shared-memory-based or kernel-assisted mechanisms thereby performing poorly when compared to their MPI counterparts. Further, the lack in performance of collective operations in OpenSHMEM will also affect the performance of parallel applications. Hence, a transparent, light-weight mechanism to map the collective operations in OpenSHMEM to their MPI counterparts holds much promise to improve the performance of collective operations. Further, this allows OpenSHMEM collectives to directly leverage the entire gamut of designs that are available in high performance MPI implementations.

In this paper, we explore the challenges associated with improving the performance of collective operations in OpenSHMEM. We propose a high-performance, light-weight caching mechanism to map the primitives in OpenSHMEM to those in MPI, thereby allowing OpenSHMEM implementations to directly leverage the advanced designs that are available in MPI libraries. We evaluate our proposed designs through various microbenchmarks and application kernels, such as Graph500 and 2D-Heat, at large scales. Based on our experimental evaluations, 2D-Heat performance is improved by around 7% and 29% at 512 processes, and Graph500 performance is improved by 57% and 82% at 4,096 processes over existing linear and tree-based algorithms.

To summarize, we address the following important problems:

1. What are the fundamental limitations that affect the performance of collective communication primitives in OpenSHMEM implementations?

2. Can we improve the performance of collective operations in OpenSHMEM through efficiently mapping them on to collective operations in high performance MPI libraries?
3. What are the potential trade-offs of such an approach; and, can we design a light-weight, transparent, and scalable interface to seamlessly improve the performance of common OpenSHMEM collectives through leveraging MPI-level designs?
4. Finally, what are performance benefits of our proposed approach across various micro-benchmarks and hybrid OpenSHMEM applications?

2 Background

2.1 PGAS models and OpenSHMEM:

In Partitioned Global Address Space (PGAS) programming models, each Processing Element (PE) has access to its own private local memory and a global shared memory space. The locality of the global shared memory is well defined. Such a model allows for better programmability through a simple shared memory abstraction while ensuring performance by exposing data and thread locality. SHMEM (SHared MEMory) [29] is a library-based approach to realize the PGAS model and offers one-sided point-to-point communication operations, along with collective and synchronization primitives. SHMEM also offers primitives for atomic operations, managing memory, and locks. There are several implementations of the SHMEM model that are customized for different platforms. However, these implementations are not portable due to minor variations in the API and semantics. OpenSHMEM [25] aims to create a new, open specification to standardize the SHMEM model to achieve performance, programmability, and portability.

2.2 Message Passing Interface:

MPI has been the dominant parallel programming model in the high performance computing domain for the past couple of decades. It has been widely ported and several open-source implementations have been made available. It has also achieved very good performance and scalability. MVAPICH2 [21] is a high-performance implementation of the MPI-2 standard on InfiniBand, iWARP, and RoCE (RDMA over Converged Ethernet). It is currently used by more than 2,070 organizations in 70 countries worldwide.

2.3 Collective Operations in MVAPICH2:

MVAPICH2 uses state-of-the-art designs, such as shared-memory-based multi-core aware designs, kernel-based zero-copy intra-node designs, and InfiniBand hardware-multicast-based designs, to improve the latency of blocking collective operations [18, 15, 30]. In these designs, the processes that are within a compute node are grouped within a "shared memory communicator." One process per node is designated as a leader and participates in a "leader communicator" that contains leaders from all nodes. The collective operations are scheduled across these communicators to achieve low communication latency. MVAPICH2 uses such a design to implement various collectives – broadcast, reduce, allreduce, scatter, gather, and barrier. The inter-leader phases can be implemented through standard point-to-point based algorithms and can be optimized using techniques like hardware-multicast, while the intra-node exchanges are implemented via shared-memory, or kernel-assisted approaches.

2.4 Unified Communication Runtime for MPI and PGAS Models:

As discussed in Section 1, it is critical to design communication runtimes that offer the best performance and scalability while supporting the use of hybrid programming models like MPI and PGAS. MVAPICH2-X [22] library, based on the Unified Communication Runtime (UCR), provides a unified and high performance communication runtime and currently supports MPI (+ OpenMP), UPC and OpenSHMEM models, and a combination of these models [11, 10]. The architecture of MVAPICH2-X is shown in Figure 1. This facilitates hybrid programming models without having the overheads of separate runtimes and their inter-operation. UCR and hence MVAPICH2-X draws from the design of MVAPICH and MVAPICH2 [21] MPI software stacks that have been optimized and widely adopted on large scale, high-performance clusters. MVAPICH2/MVAPICH2-X offer highly optimized MPI collective communication as described in Section 2.3. However, the current version (v1.9) of MVAPICH2-X does not take advantage of the optimized collective communications for OpenSHMEM. This involves several challenges as outlined in Section 1. This paper targets and addresses several of these challenges.

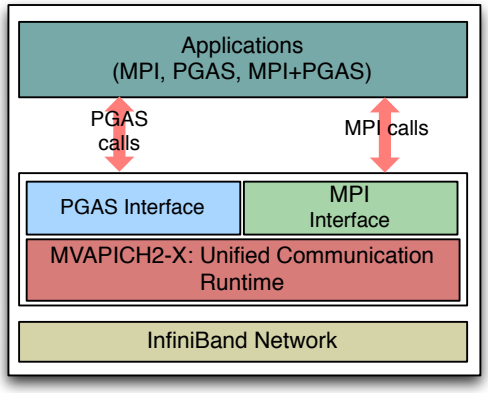


Figure 1: MVAPICH2-X: Unified Communication Runtime for MPI and PGAS

2.5 Non-collective Communicator Creation in MPI-3

MPI-3 defines a new communicator management routine for non-collective communicator creation. The syntax of the routine is as follows:
`MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm *newcomm);`
Here, the `comm` is an input parameter, which is a superset of processes in sub-group. This call requires only processes that belong to the group to participate in the communicator creation operation. Hence, this operation performs much better than the conventional MPI communicator constructor function `MPI_Comm_create` [7]. We use this MPI-3 interface for creating non-collective communicators in our design.

3 Improving Collective Communication in OpenSHMEM

This section presents the design details of implementing OpenSHMEM collectives over MPI collectives. We start this section by describing the various collectives in OpenSHMEM, followed by a discussion on the different set of challenges associated with this approach, and finally present our proposed design details.

The collective operations defined in OpenSHMEM specification v1.0 are - `shmem_barrier`, `shmem_collect`, `shmem_broadcast` and `shmem_reduce` operations. The equivalent of these operations in MPI are `MPI_Barrier`, `MPI_AllGather` and `MPI_AllReduce`. Thus, we can state that OpenSHMEM collective operations are a subset of those defined in MPI. However, we cannot directly map OpenSHMEM collective operations over MPI collectives. The challenges are explained in following section.

3.1 Design Challenges

3.1.1 Difference in Specifying Participating Processes

MPI uses a high-level communicator object to identify the group of processes that are participating in a collective communication operation. An MPI implementation may define special pre-defined communicators during MPI_Init and they remain valid for the duration of the parallel job. One such pre-defined communicator is the `MPI_COMM_WORLD` that includes all the MPI processes in the parallel job. During the course of the parallel job, new communicators can be constructed to correspond to sub-groups of processes and to define the scope of communication operations. As discussed in Section 2.3, MPI libraries create sub-communicators for each communicator object created by the application and utilize them to improve the performance of collective operations. On the other hand, in OpenSHMEM, there is no notion of a communicator. Each collective call in OpenSHMEM specifies `start_PE`, `stride_PE`, and `size` parameters. An OpenSHMEM implementation uses these parameters to dynamically define the scope of a collective operation and identify the set of processes that are participating in the specific collective operation. Hence, an OpenSHMEM implementation is required to dynamically create logical communication structures to implement collective operations, depending on the set of parameters defined by the application. Owing to these factors, state-of-the-art OpenSHMEM implementations do not utilize advanced multi-core-aware designs to optimize the performance of collective operations.

3.1.2 Expensive Communicator Creation

A simple way to map OpenSHMEM's collective operations to MPI collectives is to create a new MPI communicator for each OpenSHMEM collective operation. If an application is utilizing a unified communication library, such as, MVAPICH2-X, it can directly invoke the corresponding MPI collective operation with such a communicator. We note that such an operation can be performed transparently within the OpenSHMEM implementation, requiring no modifications to an existing OpenSHMEM application. While such a simplistic design may allow an OpenSHMEM implementation to directly utilize MPI-level designs, it may not always deliver the best communication latency. This is primarily because communicator creation in MPI is an expensive operation. These routines typically involve a collective communication operation between all the participating processes to generate context-ids. Further, the OpenSHMEM implementation must carefully release the resources allocated for such communicators, in order to prevent the MPI library from running out of internal resources. Hence, it is critical to design a light-weight interface to allow OpenSHMEM implementations to seamlessly utilize the entire range of high performance designs that are available in MPI implementations.

3.2 Detailed Design

We take on these challenges and come up with a design with light weight *Communicator Creator* and a *Communicator Cache*. The overall design is depicted in Figure 2. For every OpenSHMEM collective call, we propose to first check if it defines a process group to include all the processes in the parallel job. In this case, the OpenSHMEM collective operation can be directly mapped to an equivalent MPI collective routine, with the `MPI_COMM_WORLD` communicator. If the process group defined by the OpenSHMEM application does not correspond to `MPI_COMM_WORLD`, we propose to maintain a cache of communicators, *Communicator Cache*. Our design performs a look-up operation to identify a matching communicator. If our communication runtime has already created such a communicator, we consider this as a "cache-hit" and we directly re-use

this communicator. However, if the process group does not correspond to any cached communicator, we treat this as a "cache-miss" and we create a new communicator using *Communicator Creator* and we cache the newly created communicator in *Communicator Cache*. Our *Communicator Creator* component relies on the non-collective communicator constructor (Section 2.5), to create the communicator in *O(log N)* time (where N is the number of participating processes). The design details of communicator creator and cache are explained in following sections.

3.2.1 Communicator Creator

As discussed in Section 2.5, the routines provided by MPI for communicator creation are collective over an existing parent communicator. MPI-3 has proposed a routine for creating non-collective communicator. We base our communicator creation based on this communicator creation routine. The algorithm is collective only on processes that are members of group, and the definition of the group must be identical across all the MPI processes. The complexity of this algorithm is *log(N)*, where N is the number of participating processes.

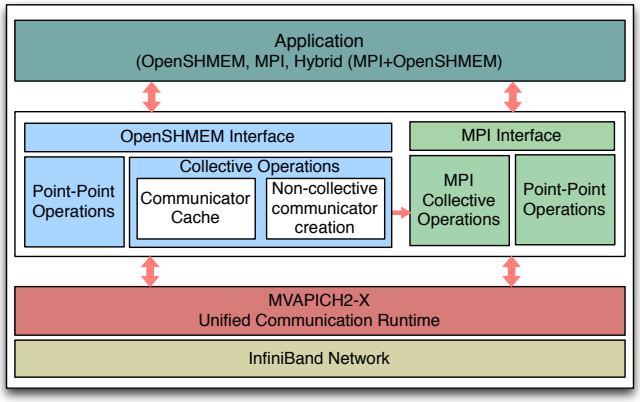


Figure 2: Proposed Design of OpenSHMEM Collectives

3.2.2 Communicator Cache

Even though the algorithm takes *log(N)* steps for communicator creation, it is not advisable to create communicator during every collective call. We implemented a Least Recently Used (LRU) based cache for caching communicator. The communicators are held in this cache. When an application invokes an OpenSHMEM collective on a process group that matches a cached-communicator, we directly re-use it. The size of cache is configurable. Depending on the utilization pattern of the cached-communicators, our design transparently destroys the least recently used communicator and releases all its resources. Finally, when the application terminates, all the cached communicators are freed.

4 Experimental Evaluation

In this Section, we describe our experimental test-bed and discuss our evaluations. We study the performance characteristics of collective operations with our proposed designs, across various micro-benchmarks, pure OpenSHMEM, and hybrid MPI+OpenSHMEM applications.

4.1 Experiment Setup

We used two clusters for performance evaluations.
Cluster A: This cluster consists of 144 compute nodes with Intel Westmere series of processors using Xeon Dual quad-core processor nodes operating at 2.67 GHz with 12 GB RAM. Each

node is equipped with MT26428 QDR ConnectX HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces. The operating system used is Red Hat Enterprise Linux Server release 6.3 (Santiago), with kernel version 2.6.32-71.el6 and OpenFabrics version 1.5.3-3.
Cluster B: This cluster (TACC Stampede [31]) is equipped with compute nodes with Intel Sandybridge series of processors using Xeon dual eight-core sockets, operating at 2.70 GHz with 32 GB RAM. Each node is equipped with MT4099 FDR ConnectX HCAs (54 Gbps data rate) with PCI-Ex Gen3 interfaces. The operating system used is CentOS release 6.3, with kernel version 2.6.32-279.el6 and OpenFabrics version 1.5.4.1. Even though this system has large number of cores, we were able to gain access to only 8,192 cores for running experiments for this paper.

For all the experiments, we have used MVAPICH2-X OpenSHMEM based on OpenSHMEM version 1.0vd [25]. We used Graph500 v2.1.4 in our experiment evaluations. For all microbenchmark evaluations, we report results that are averaged across 1,000 iterations and three different runs to eliminate experimental errors. We used Cluster A for micro-benchmark and 2D-Heat application kernel evaluation and Cluster B for Graph500 evaluations.

4.2 MicroBenchmark Evaluations

In this section, we compare the performance of various collective operations in OpenSHMEM, across various implementations and design choices, with a varying number of processes. Specifically, we are interested in exploring the differences between OpenSHMEM's linear and tree-based approaches for implementing collective operations and our proposed approach that allows us to map OpenSHMEM's collectives to those in MPI. In the following figures, we refer to our proposed designs as "OSHM-Hybrid". We also include a comparison with the latency of the corresponding MPI collective operation.

In Figures 3(a), (b), and (c), we study the performance of the OpenSHMEM's `shmem_collect` operation, across varying number of processes. We note that the current version of OpenSHMEM reference implementation [25] does not include tree-based algorithm for `shmem_collect`. Hence, we compare OpenSHMEM's default Linear version (denoted as "OSHM-Linear") with our proposed Hybrid design and MVAPICH2's default implementation of the corresponding `MPI_Allgather` collective. We note that for all the three system sizes, our proposed designs deliver significant benefits, up to 1000X times better than OpenSHMEM's default Linear implementation. We also note that our design offers the same communication performance as the MPI implementation. Hence, we note that our proposed mapping mechanism introduces very little overheads, while allowing OpenSHMEM collectives to seamlessly leverage the efficient designs that are available in MVAPICH2.

Similarly, in Figures 4 (a), (b), and (c), we study the performance characteristics of different design alternatives for the `shmem_broadcast` collective operation, with varying number of processes. For this collective, we also include a comparison with the Tree-based implementation (denoted as "OSHM-Tree") that is available in OpenSHMEM, along with the Linear implementation, our proposed Hybrid version, and the default implementation in the MVAPICH2 library. We observe that the OpenSHMEM's tree-based design outperforms the Linear design. This is primarily because the tree implementation can allow the entire operation to complete in a maximum of $log(N)$ steps, with N OpenSHMEM processes. However, in the Linear scheme, the root of the broadcast needs to perform N steps to individually transfer the data to every other process. We note that our proposed Hybrid design performs about 10X better than the tree-based implementation of broadcast in OpenSHMEM and is comparable to the performance of the default implementation of `MPI_Bcast` in MVAPICH2. These performance benefits mainly

arise from the shared-memory based, hierarchical designs that are used in the MVAPICH2 library. Since such advanced designs are not available in pure OpenSHMEM implementations, our designs lead to significant performance benefits.

Finally, in Figures 5(a), (b), and (c), we perform a similar comparison for OpenSHMEM's `shmem_reduce` collective operation. We note that our proposed Hybrid design outperforms OpenSHMEM's default implementation by a factor of 100 and does not add any additional overheads, when compared to the default MPI implementation of Allreduce.

In Figures 6, we compare the performance of the `shmem_broadcast` operation with a different process sub-group, instead of allowing all the OpenSHMEM processes to participate in the collective operation. This benchmark is designed to demonstrate the performance of various design choices, with slightly varying process groups. As a simple variant, we allow only the processes with even ranks to participate in the collective operation. In this version of the benchmark, we note the performance characteristics are fairly similar to those discussed in Figures 4(a), (b), and (c). This observation is primarily because once the new process sub-group has been formed, the new communicator is cached within the communication library and it can be re-used for any subsequent operation with the same process group.

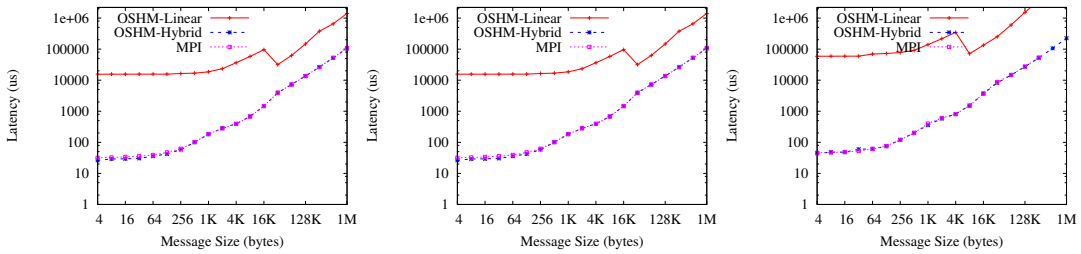


Figure 3: Collect Performance Comparison (All Processes) (a) 128 Processes, (b) 256 Processes, and (c) 512 Processes

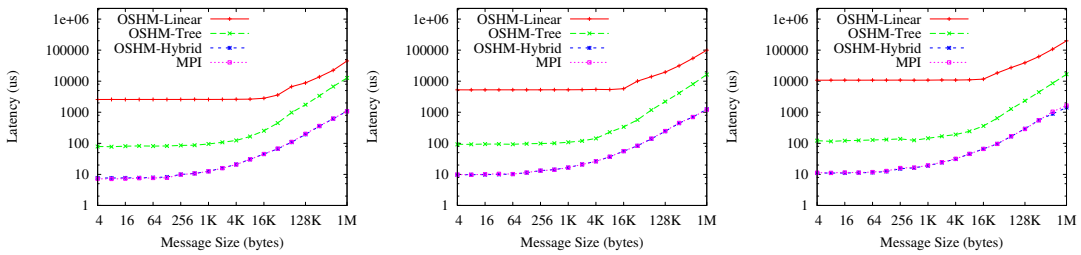


Figure 4: Broadcast Performance Comparison (All Processes) (a) 128 Processes, (b) 256 Processes, and (c) 512 Processes

4.3 OpenSHMEM Application Evaluation

We consider two application kernels — 2D-Heat and Graph500 [12] — for performance evaluation. The 2D-Heat application kernel is available in OpenSHMEM v.1.0 release. This benchmark uses Gauss-Seidel method for modeling 2D heat conduction. Gauss-Seidel kernel is repeated until the standard deviation between adjacent 2D matrices is less than a predefined convergence value. In the Gauss-Seidel kernel, data transfer between adjacent processes

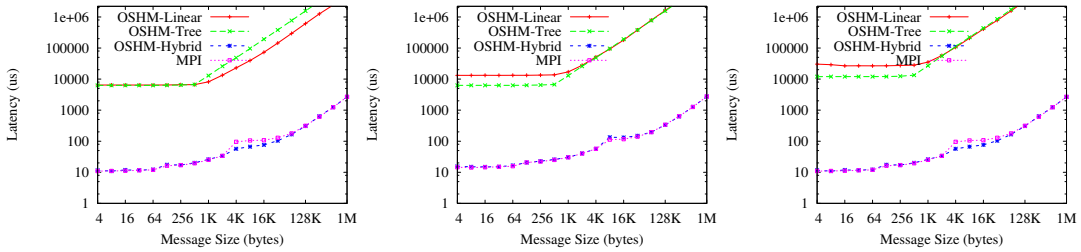


Figure 5: Allreduce Performance Comparison (All Processes) (a) 128 Processes, (b) 256 Processes, and (c) 512 Processes

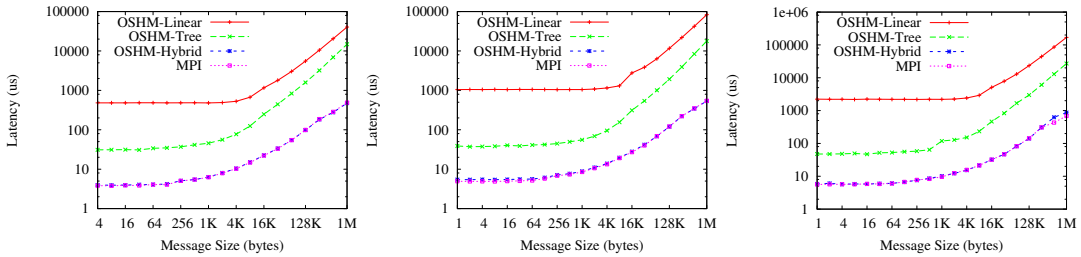


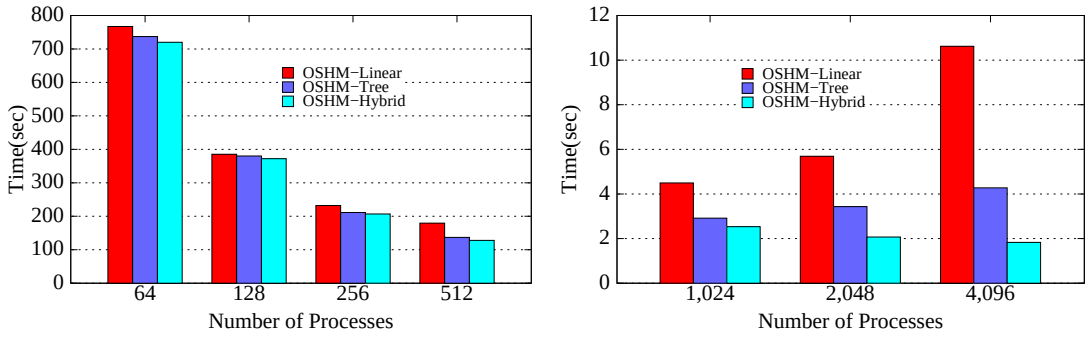
Figure 6: Broadcast Performance Comparison (Even Processes) (a) 128 Processes, (b) 256 Processes, and (c) 512 Processes

are performed using `shmem_float_put` and synchronization between stages are achieved using `shmem_barrier_all` calls. After every iteration of Gauss-Seidel kernel, sum of squares is calculated using `shmem_float_sum_to_all` calls. PE 0 calculates the square root and is broadcasted to all PE's using `shmem_broadcast64`. Figure 7(a) shows the performance results of 2D-Heat Transfer Modeling benchmark. We used an input matrix of size 8 K x 8 K for this experiment. The benchmark was run for different number of processes, as shown in the figure. As it can be noted from the results, our design based on MPI collectives performs better than both linear and tree based collective designs, for all scales. Moreover, the performance benefits improve as we increase the system size. With 512 processes, the execution times were 179.6, 136.9, and 127.0 seconds for linear, tree, and hybrid designs, respectively. This is about 7% and 29% improvement, compared to tree and linear collective designs.

The second application kernel in our performance evaluation is Graph500 benchmark. The Graph500 Benchmark Specification [32] is a new set of benchmarks to evaluate scalability of supercomputing clusters in the context of data-intensive applications. We use Concurrent Search benchmark kernel of Graph500 suite in our application evaluation. It is basically a Breadth First Search (BFS) traversal benchmark based on level synchronized BFS traversal algorithm. In the benchmark, each participating process keeps two queues — 'CurrQueue' and 'NewQueue'. In each level, vertices in CurrQueue are traversed, and the newly visited vertices are put into NewQueue. At the end of each level, the queues are swapped. When the queues are empty at all the participating processes, the algorithm terminates. We modify this benchmark to use the collective reduction operation `shmem_longlong_sum_to_all` for identifying if the queues are empty. The number of times the collective operation gets called depends on the size of the graph and problem scale size. We used an input graph of size 512 million edges and eight billion vertices. We used cluster B for Graph500 evaluation.

The performance results of Graph500 hybrid benchmark are shown in Figure 7(b). The new

design based on MPI collectives improves the performance significantly. For 4,096 processes, the new design takes about 1.81 seconds where as the linear and tree based designs take 10.6 and 4.2 seconds, respectively. This is about 82% and 57% improvement, respectively.



(a) 2D Heat Transfer Modeling (8Kx8K Input)

(b) Graph 500

Figure 7: Performance of OpenSHMEM Applications

For both the applications, the new design of OpenSHMEM collectives based on MPI collectives performs better for all system sizes. The performance benefits improve as we increase the number of processes.

5 Related Work

There have been several SHMEM variants after Cray Research introduced it in Cray T3D platform [5]. Some of the major implementations are QSHMEM [27], SGI-SHMEM [29], GP-SHMEM [13], HP-SHMEM, and IBM-SHMEM. MPI implementation of SHMEM [2] are also available. Some of these implementations are still in use today. However, every variant has its own spin off the SHMEM API semantics. With the recent OpenSHMEM effort [4] in unifying SHMEM specifications, a renewed interest has been observed in developing high-performance implementation of the OpenSHMEM specification. Sandia National Laboratories recently developed OpenSHMEM over Portals network programming interface [1]. Portals is a low-level network data movement layer and programming interface to support higher-level one-sided and two-sided interfaces. Latest Portals 4 [28] has included support for PGAS programming languages. Brightwell et. al proposed an intra-node implementation of OpenSHMEM, that uses operating system virtual address space mapping capabilities to provide efficient intra-node operations [3]. Potluri et. al. proposed a high-performance hybrid intranode runtime for OpenSHMEM by taking advantage of shared-memory-based, kernel-based and network-loopback-based techniques [26].

Yoon et. al developed portable OpenSHMEM library called GSHMEM [33]. GSHMEM employs GASNet communication middleware from UC Berkeley and is based on the v1.0 draft of the OpenSHMEM specification. OpenSHMEM reference implementation (OpenSHMEMv1.0) [25] also uses GASNet as the communication subsystem. Using GASNet-InfiniBand conduit, these OpenSHMEM implementations enable OpenSHMEM communication on InfiniBand networks. Several papers mentioned above include a discussion on optimized implementation of the OpenSHMEM collective operations, some proposing optimized algorithms and some taking advantage of the new communication channels introduced [3, 26]. Dinan et. al. proposed

the non-collective communicator creation in MPI [7] and highlighted that such algorithms can be used for PGAS models.

Nishtala et. al, have proposed optimizations to improve the performance of UPC collectives [24, 23]. In this paper, we address the challenges involved in improving the performance of collective operations in OpenSHMEM. We propose light-weight, transparent designs to map collective operations in OpenSHMEM directly to their equivalent collective operations in MPI. Since state-of-the-art MPI libraries rely on a set of highly optimized designs to implement collective operations, our work potentially allows OpenSHMEM implementations to seamlessly leverage the designs that are available in MPI implementations.

6 Conclusion and Future Work

In this paper, we explored the challenges associated with improving the performance of collective operations in OpenSHMEM using MPI collectives. We proposed a high-performance, light-weight caching mechanism to map the primitives in OpenSHMEM to those in MPI, thereby allowing OpenSHMEM implementations to directly leverage the advanced designs that are available in MPI libraries. We evaluated our proposed designs through various microbenchmarks and application kernels - Graph500 and 2D-Heat, at large scales. Our experimental evaluations reveal that 2D-Heat performance is improved by around 7% and 29% at 512 processes, and Graph500 performance is improved by 57% and 82% at 4,096 processes over existing linear and tree based algorithms. We plan to design a light weight communicator mechanism at the runtime layer, so that any programming model can leverage it.

References

[1] B. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, and K. Underwood. Enhanced Support for OpenSHMEM Communication in Portals. In *IEEE Annual Symposium on High Performance Interconnects*, 2011.

[2] Ron Brightwell. A New MPI Implementation For Cray SHMEM. In *In PVM/MPI*, 2004.

[3] Ron Brightwell and Kevin Pedretti. An Intra-Node Implementation of OpenSHMEM Using Virtual Address Space Mapping. In *PGAS*, 2011.

[4] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *PGAS*, 2010.

[5] Cray, Inc. Man Page Collection: Shared Memory Access (SHMEM). (S-2383-23).

[6] D. Buntinas, B. Goglin, D. Goodell, G. Mercier and S. Moreaud. Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis. *ICPP '09*, pages 462–469.

[7] James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, and Abhinav Vishnu. Noncollective Communicator Creation in MPI. EuroMPI '11, 2011.

[8] G. Almasi, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E Moreira, Steinmacher-Burow, B. and Y. Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *ICS '05*, pages 253–262, 2005.

[9] J. Vetter and F. Mueller. Communication Characteristics of Large-scale Scientific Applications for Contemporary Cluster Architectures. *J. Parallel Distrib. Comput.*, 63:853–865, 2003.

[10] J. Jose, K. Kandalla, Miao Luo, and D.K. Panda. Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation. In *Parallel Processing (ICPP), 2012 41st International Conference on*, 2012.

[11] J. Jose, M. Luo, S. Sur, and D. K. Panda. Unifying UPC and MPI Runtimes: Experience with MVAPICH. In *PGAS*, 2010.

[12] Jithin Jose, Karen Tomko Sreeram Potluri, and Dhabaleswar. K. Panda. Designing Scalable Graph500 Benchmark with Hybrid MPI+OpenSHMEM Programming Models. International Supercomputing Conference (ISC), 2013.

[13] K. Parzyszek. Generalized Portable Shmem Library for High Performance Computing. In *Doctoral Thesis. UMI Order Number: AAI3105098.*, Iowa State University, 2003.

[14] Shoaib Kamil, John Shalf, Leonid Oliker, and David Skinner. Understanding ultra-scale application communication requirements. In *in Proceedings of the 2005 IEEE International Symposium on Workload Characterization (IISWC)*, pages 178–187, 2005.

[15] Krishna Kandalla, Hari Subramoni, Gopal Santhanaraman, Matthew Koop, and Dhabaleswar K. Panda. Designing Multi-leader-based Allgather Algorithms for Multi-core clusters. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2009.

[16] Sameer Kumar, Yogish Sabharwal, Rahul Garg, and Philip Heidelberger. Optimization of All-to-All Communication on the Blue Gene/L Supercomputer. In *Proc. of 37th Int. Conf. on Parallel Processing*, pages 320–329, 2008.

[17] Teng Ma, George Bosilca, Aurelien Bouteiller, and Jack J Dongarra. Kernel-Assisted and Topology-Aware MPI Collective Communications on Multi-Core/Many-Core Platforms. *Journal of Parallel and Distributed Computing*, 2013.

[18] A. Mamidala, R. Kumar, D. De, and D. K.Panda. MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics. In *8th IEEE International Symposium on Cluster Computing and the Grid, 2008, Lyon*, pages 130–137, May 2008.

[19] A. R. Mamidala, A. Vishnu, and D. K. Panda. Efficient Shared Memory and RDMA Based Design for MPI-Allgather over InfiniBand. In *13th European PVM/MPI User's Group Meeting, Bonn, Germany, September 17-20, 2006, Vol. 4192*, 2006.

[20] MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing*, 1993.

[21] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/.

[22] MVAPICH2-X: Unified MPI+PGAS Communication Runtime over OpenFabrics/Gen2 for Exascale Systems. http://mvapich.cse.ohio-state.edu/.

[23] R. Nishtala, P. H. Hargrove, D. O. Bonachea, and K. A. Yelick. Scaling Communication-Intensive Applications on BlueGene/P Using One-Sided Communication and Overlap. In *IPDPS '09 Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009.

[24] Rajesh Nishtala and Katherine A. Yelick. Optimizing Collective Communication on Multicores. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 18–18, 2009.

[25] OpenSHMEM. http://openshmem.org/.

[26] Sreeram Potluri, Krishna Kandalla, Devendar Bureddy, Mingzhe Li, and Dhabaleswar K. Panda. Efficient Intranode Desgins for OpenSHMEM on Multicore Clusters. In *PGAS*, 2012.

[27] Quadrics Ltd. Quadrics SHMEM programming manual.

[28] Sandia National Laboratories. The Portals 4.0 Message Passing Interface. http://www.cs.sandia.gov/Portals/publications/portals40.pdf.

[29] Silicon Graphics International. SHMEM API for Parallel Programming. http://www.shmem.org/.

[30] Sayantan Sur, Uday Kumar Reddy Bondhugula, Amith Mamidala, Hyun-Wook Jin, and Dhabaleswar K. Panda. High Performance RDMA Based All-to-all Broadcast for InfiniBand Clusters. In *In the Proceedings of the 12th International Conference of High Performance Computing (HIPC), Goa, India*, 2005.

[31] TACC Stampede Cluster. www.xsede.org/resources/overview.

[32] The Graph500. http://www.graph500.org.

[33] C. Yoon, V. Aggarwal, V. Hajare, A. D. George, and M. Billingsley III. GSHMEM: A Portable Library for Lightweight, Shared-Memory, Parallel Programming. In *PGAS*, 2011.

UPC on MIC: Early Experiences with Native and Symmetric Modes *

Miao Luo, Mingzhe Li, Akshay Venkatesh, Xiaoyi Lu,
and Dhabaleswar K. (DK) Panda

Department of Computer Science and Engineering,
The Ohio State University
{luom, limin, akshay, luxi, panda}@cse.ohio-state.edu

Abstract

Intel Many Integrated Core (MIC) architecture is steadily being adopted in clusters owing to its high compute throughput and power efficiency. The current generation MIC coprocessor, Xeon Phi, provides a highly multi-threaded environment with support for multiple programming models. While regular programming models such as MPI/OpenMP have started utilizing systems with MIC coprocessors, it is still not clear whether PGAS models can easily adopt and fully utilize such systems. In this paper, we discuss several ways of running UPC applications on the MIC architecture under Native/Symmetric programming mode. These methods include the choice of process-based or thread-based UPC runtime for *native* mode and different communication channels between MIC and host for *symmetric* mode. We propose a thread-based UPC runtime with an improved "leader-to-all" connection scheme over InfiniBand and SCIF [3] through multi-endpoint support. For the *native* mode, we evaluate point-to-point and collective micro-benchmarks, Global Array Random Access, UTS and NAS benchmarks. For the *symmetric* mode, we evaluate the communication performance between host and MIC within a single node. Through our evaluations, we explore the effects of scaling UPC threads on the MIC and also highlight the bottlenecks (up to 10X degradation) involved in UPC communication routines arising from the per-core processing and memory limitations on the MIC. To the best of our knowledge, this is the first paper that evaluates UPC programming model on MIC systems.

1 Introduction

Over the past decade, the field of High Performance Computing has witnessed a steady growth in the density of compute cores available in nodes. The advent of accelerators and coprocessors has pushed these boundaries further. The Intel Many Integrated Core (MIC) architecture (known as Xeon Phi) provides higher degrees of parallelism with 240+ threads on 60+ low powered cores on a single PCIe card. Such developments have made it increasingly common for High Performance Computing clusters to use these coprocessors that pack a theoretical 1 Teraflop double precision compute throughput per card. This trend is evident in the recent TOP500 list [4], where systems equipped with MIC coprocessors have risen from seven to twelve in the past six months (from November 2012 to June 2013). In fact, Tianhe-2 [7], which ranks first in the current Top500 list, is composed of 32,000 Intel Ivy Bridge Xeon Sockets and 48,000 Xeon Phi boards (total of 3,120,000 cores).

The Intel Xeon Phi [2], providing x86 compatibility, runs a Linux operating system and offers a highly flexible usage model for application developers. MIC supports the use of several popular

*This research is supported in part by National Science Foundation grants #OCI-0926691, #OCI-1148371 and #CCF-1213084.

programming models including MPI, OpenMP, Thread Building Blocks, and others that are used on multi-core architectures. This dramatically reduces the effort of porting applications developed for multi-core systems onto the Xeon Phi.

High end systems have been deployed with high performance interconnects such as InfiniBand [13]. The latest TOP500 list shows 41% of the supercomputing systems rely on InfiniBand as their network interconnects. InfiniBand offers Remote Direct Memory Access (RDMA) features to deliver low latency and high bandwidth to HPC systems and scientific applications. Intel MPSS [1] enables InfiniBand-based communication from the MIC to local and remote host processors as well as local and remote MICs. This allows MIC-clusters to leverage on popular distributed memory programming models such as Message Passing Interface (MPI).

The unique environment, that emerging hybrid many-core systems with high-performance interconnects, poses several optimization challenges for HPC applications. Recent studies [18, 19, 11] have analyzed the challenges and effects of running regular programming models such as MPI/OpenMP on these emerging heterogeneous systems. However, the irregular programming models, such as PGAS and its implementations like Unified Parallel C (UPC) [22], are still not well investigated on the platforms with Intel MIC coprocessors. As high end clusters equipped with InfiniBand networks and MIC coprocessors gain attention, the broad question remains: *How to optimally run PGAS implementations, such as UPC, on such MIC clusters and what will be the corresponding performance characteristics?*

2 Motivation

2.1 Problem Statement

On the Intel MIC, applications are usually run in one of the following modes:

1. ***Native***: MIC can be used in a *native* many-core mode where the application runs only on the coprocessor.

2. ***Offload***: MIC can be also used in an *offload* accelerator mode where the application runs only on the host and offloads compute-intensive parts of code to the coprocessor.

3. ***Symmetric***: A *symmetric* mode is also offered on MIC where the application can be launched on both the coprocessor and the host. This mode provides maximum control in the way applications are launched and allows application developers to take full advantage of the resources offered by the host and coprocessor.

Several previous works [18, 19, 6] have explored running MPI applications on MIC. With both the *native* mode and the *symmetric* mode, the limited memory on the MIC places constraints on the number of MPI processes that can run on it and hence can be prohibitive towards fully utilizing the hardware capabilities available. In order to overcome this limitation, hybrid programming models of MPI and OpenMP can be deployed for less process count and memory footprint; and, allow for light-weight threads to fully subscribe all the cores on the MIC. However, this approach involves more programming efforts to modify MPI applications with OpenMP directives. Hence the availability of a highly multi-threaded environment and the need for high programming productivity makes PGAS implementations such as UPC a more natural fit for the MIC.

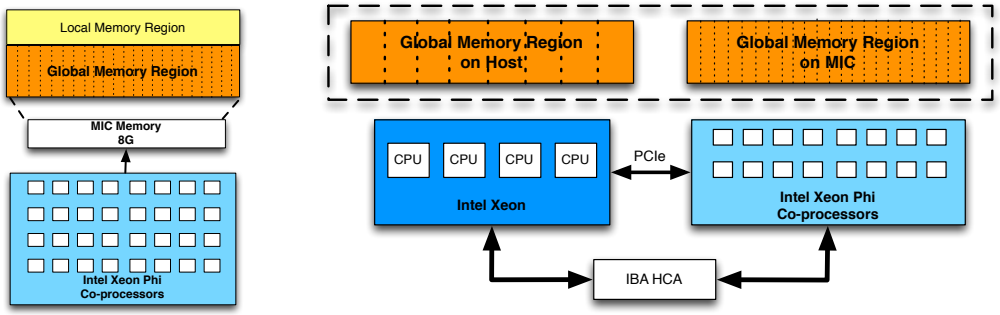
The *offload* mode allows programmers to use compiler directives to offload regions of computation that may benefit from acceleration on the coprocessor. However, in this mode, the

Coprocessor Offload Engine (COE) manages all memory operations with coprocessor memory, and hence, any data residing on the coprocessor must be first copied on to the host through COE if there is an intent to send that data out of the node. This increases the memory copy overheads on applications. Furthermore, programmers are not exempted from changing the source code of applications for marking the regions of computation to be offloaded. Also, for computations characterized by irregular communication patterns, it is hard to mark the regions of computation to be offloaded due to the irregular computation feature. Therefore, this paper mainly explores and discusses the situations of evaluating the performance characteristics of UPC runtime and running UPC applications under *native* and *symmetric* modes. Even for these two modes, we still need to address and investigate additional problems, as outlined below:

1. UPC threads can be mapped to either OS process or thread. The process-based runtime and thread-based runtime rely on different schemes for intra-node communication. Taking the new features of MIC into consideration, what are the performance differences between these two runtime schemes?

2. As shown in Figure 1(a), the limited memory on the MIC leaves dozens of UPC threads sharing a small global memory region in the *native* mode. What is the performance impact on UPC applications of this kind of many-core architecture with a small memory space?

3. As shown in Figure 1(b), in the *symmetric* mode, the host memory space and coprocessor memory space will be integrated as a global memory region for UPC threads running on both host and coprocessor. What are the performance characteristics of the communication between host and MIC?



(a) Mapping Global Shared Memory Region on MIC for *native* mode

(b) Connect MIC and host in *symmetric* mode

Figure 1: UPC on MIC: *native* and *symmetric* modes

2.2 Contributions

This paper addresses all of above issues and shares our early experiences of UPC on the MIC under *native* and *symmetric* modes. This paper makes the following key contributions:

1. We compare the pros and cons of process-based and thread-based UPC runtime implementations and show that thread-based runtime is a better alternative for UPC on MIC in the *native* mode.

2. We propose a new connection mode for thread-based runtime with multi-endpoint support, in order to address the communication issue for host-to-MIC and MIC-to-host in *symmetric* mode. The new connection mode can reduce the number of required connections from $O(N^2)$ to $O(N)$ while keeping the shared array access between host and MIC efficient.

3. We present and analyze the proposed thread-based runtime with new connection mode for intra-MIC, host-to-MIC, and MIC-to-host separately. We also evaluate the impact of MIC on UPC benchmarks with different communication patterns such as Random Access, NAS, and UTS.

To the best of our knowledge, this is the first paper that evaluates UPC programming model on MIC systems. The rest of the paper is organized as follows. Section 3 discusses some background on UPC and InfiniBand. We further discuss alternatives for running UPC on MIC in Section 4. Section 5 gives our experimental results. Section 6 lists the related work. Finally, we conclude the paper in Section 7.

3 Background

In this Section, we provide an overview of UPC and InfiniBand architectures.

3.1 UPC

Unified Parallel C (UPC) [22] is an emerging parallel programming language that aims to increase programmer productivity and application performance by introducing parallel programming and remote memory access constructs in the language. There are several implementations of the UPC language. The Berkeley UPC implementation [15] is a popular open-source implementation of UPC. IBM and Cray also distribute their own versions of UPC implementations specifically optimized for their platforms.

GASNet interface is utilized by Berkeley UPC for memory access across network. It consists of Core APIs and Extended APIs [9]. The Core API interface is a narrow interface based on the Active Message paradigm. Extended APIs provide a rich, expressive, and flexible interface that provides medium and high-level operations on remote memory and collective operations. GASNet supports different network conduits, viz., ibv (OpenIB/OpenFabrics IB Verbs), udp (UDP), lapi (IBM LAPI) [12], mpi (MPI), etc. In previous work [14], we proposed a new high performance Unified Communication Runtime (UCR) for InfiniBand network support to Berkeley UPC through GASNet interface. A network endpoint contention free UCR with multi-endpoint support is proposed in [16].

3.2 InfiniBand Architecture

InfiniBand network [13] has been widely used in the high performance computing systems. In June 2013, there are 41% systems in TOP500 list using InfiniBand as the primary network. The most powerful feature in InfiniBand is the Remote Direct Memory Access (RDMA). The RDMA-based communication can be processed without any remote host processor involvement.

The basic RDMA operations are RDMA write and RDMA read, which are used in the UPC runtime system to implement `upc_memput` and `upc_memget`, respectively.

4 Alternatives for Running UPC on MIC

4.1 UPC Runtime: Process-based or Thread-based?

UPC *thread* refers to an instance of execution for UPC applications. Depending on different implementations of UPC runtime, each UPC thread can be mapped to either an OS process or an OS thread. While OS threads share the entire address space, processes can only share certain memory regions. Thus, in existing process-based runtime, the intra-node memory access between two UPC threads has to be realized through one of the two shared memory based schemes. The first scheme is copy-via-shared-memory. In this scheme, shared memory performs as intermediate buffer. The owner of the source copies the data from source buffer to a temporary buffer that has been mapped to shared memory. Then the owner of the destination copies the data from the temporary buffer to the destination buffer. There are three disadvantages of the copy-via-shared-memory scheme: overhead from extra copy, extra memory footprint for intermediate buffer, and extra synchronization between source and destination processes. Many-core systems such as MIC have larger number of cores and limited memory compared to normal, multi-core systems. The copy-via-shared-memory scheme thus suffers higher overhead on MIC. Furthermore, the synchronization between source and destination is also against the one-sided native feature of UPC programming language.

The second scheme utilized by process-based runtime is shared-memory-mapping. The Berkeley UPC runtime offers a feature called PSHM (Inter-Process SHared Memory) [5]. PSHM provides support for UPC shared arrays using Unix System V (SysV) shared memory. The whole local part of the shared memory region, which belongs to one process, needs to be mapped into the address space of all the other processes. All other processes then can direct read/write UPC shared arrays through shared memory mapping. This removes the extra copy overhead and synchronization problems in the first scheme. However, the mapping of shared region from every process to all the other processes generates a huge amount of memory footprint in the kernel space. The number of required entries in page table is $O(N^2)$, where N equals to the number of processes. In a many-core system like MIC, with possibly more than 60 processes, a large amount of memory is consumed for the page table. In short, the problems with the two shared memory based schemes on multi-core systems are worse off on many-core systems like MIC.

On the other hand, thread-based runtime can achieve much lower latency to directly access shared array with no extra memory requirement. This is because all the UPC threads on the same node are mapped to OS threads spawned by a single OS process. As a result, the global shared array region within a single node belongs to the same address space. UPC direct memory access within a single node between different UPC threads then can be done by system memory copy functions directly from the source buffer to destination buffer. Regarding to the drawbacks of thread-based runtime, previous research [8] has reported that it suffers from bad network performance, due to sharing of network connections when hardware allows one connection per process. Our previous work [16] proves that through multi-endpoint support, thread-based runtime can achieve the same performance as a process-based runtime.

Based on the comparison between process-based and thread-based runtime implementations for intra-node communication, in the many-core system with increasing intra-node communication and limited memory space, thread-based UPC runtime with multi-endpoint design is the

right direction from both performance and scalability point of view. We thus choose to utilize thread-based runtime for the evaluations in the following sections of the paper.

4.2 Remote Memory Access Between MIC and HOST

Communications between MIC and host can go through either SCIF-based [3] channel, which is based on PCIe, or IB-based channel. Both of these channels provide remote memory access (RMA) semantics. In the context of MPI programming, it has been shown that, for this configuration, IB is suited in the small message range and SCIF is suited in the large message range [19]. For IB-based communication in the UPC process-based runtime, every process on the MIC needs to establish a connection with every process on the host and vice versa. This requires order of $N_{MIC} \times N_{HOST}$ number of connections, where N_{MIC} and N_{HOST} refer to the number of UPC threads on MIC and host, respectively. With 60+ cores on MIC, the number of connections will increase significantly for a large value of N_{MIC}. Hence the process-based runtime can place increased pressure on the limited memory residing on the coprocessor. In order to reduce the memory footprint for MIC-to-HOST communication without compromising performance, we propose a "leader-to-all" connection mode for multi-endpoint support [16] in the multi-threaded runtime.

Figure 2 shows how the "leader-to-all" connection mode works. First of all, the MIC leader thread MIC_{leader} registers a shared memory region local to the whole MIC co-processor. After the registration, all other threads on MIC can access this shared memory region just pinned-down by MIC_{leader}. The host leader thread, $HOST_{leader}$, does the same registration. After MIC_{leader} and $HOST_{leader}$ finish registration, those two leaders exchange their rkeys of the pinned-down memory. Those two rkeys are distributed to all the threads on MIC and host by the MIC_{leader} and $HOST_{leader}$, respectively. Then all the threads on MIC (host) establish connections with the remote leader thread on host (MIC). When a thread $HOST_i$ needs to access a shared array address belonging to any thread MIC_j on the MIC, thread $HOST_i$ uses the connection between itself and the thread MIC_{leader} with the distributed rkey. Through this "leader-to-all" connection mode, the number of connections between MIC and host is reduced to $N_{MIC} + N_{HOST}$.

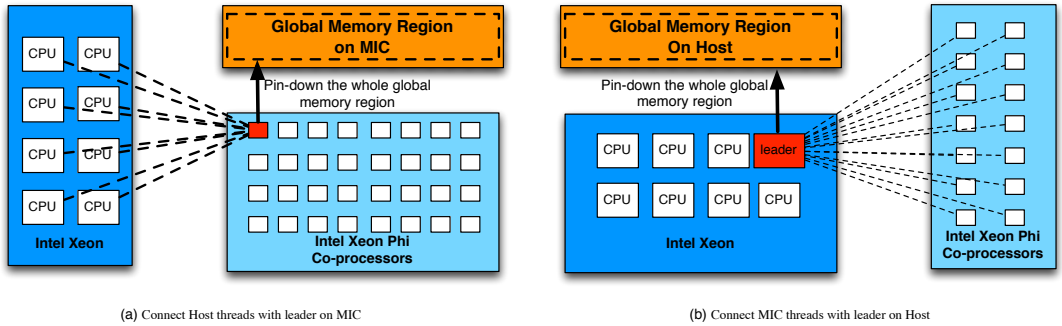


Figure 2: Leader-to-all Connection Mode

For large message communication, where SCIF-channel performance is more optimal, we reuse the "leader-to-all" connection design through IB over SCIF mechanism [17].

5 Results

In this section, we evaluate the proposed multi-threaded UPC runtime with micro-benchmarks and applications.

5.1 Experimental Setup

Our experimental environment is a dual socket node containing Intel Sandy Bridge (E5-2680) dual octa-core processors, running at 2.70GHz, a SE10P (B0-KNC) coprocessor and a Mellanox IB FDR MT4099 HCA. The host processors are running CentOS release 6.3 (Final), with kernel version 2.6.32-279.el6.x86 64. The KNC runs MPSS 4346-16 (Gold). The compute node is configured with 32GB of "host" memory with an additional 8GB of memory on the Xeon Phi coprocessor card. The Intel compiler composer xe 2013.0.079 has been used. All the micro-benchmark results are averaged over 1,000 iterations.

The UPC implementation used in the evaluation section is based on Berkely UPC v2.16.0 and the multi-endpoint UCR runtime proposed in [16]. As discussed in Section 4.1, we have all the UPC threads mapped to OS threads and "leader-to-all" connection mode is utilized to enable communication between host and MIC in *symmetric* mode.

MIC can support quad hyper threads for each core. However, when we evaluated the performance of Hyper Threading by binding more than one thread on each core, the performance drops dramatically. Thus in the following evaluations, there is only one thread on each MIC core.

5.2 Micro-benchmark Evaluations

In this section, we present the performance evaluation results of point-to-point and collective micro-benchmarks.

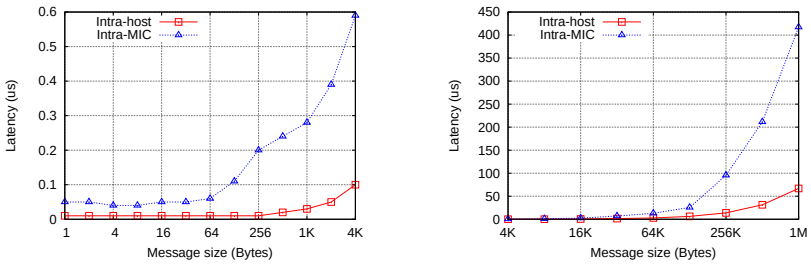
5.2.1 Native Mode Point-to-Point Evaluations

We first evaluate the *native* mode by running benchmarks purely on MIC and host, respectively. "Intra-host" refers to the results when running with *native* mode on host, where two UPC threads are all launched on host. "Intra-MIC" refers to the results with *native* mode on MIC.

In Figure 3, we compare UPC memput performance running with 2 threads on MIC or host separately. The latency of a 256 byte message size memput operation running on MIC and host are 0.2 and 0.01 μs, respectively. For large messages, a 1 MB message size latency on MIC and host are 440 and 60 μs. We observe that there is a huge gap in the performance of the memput operation on the two platforms. On the MIC, there is up to 10X memory copy overhead for a single communication pair in comparison with that on the host. This difference in performance can be attributed to the difference in performance of the *memcpy* operation as well as the frequency at which the cores run on these two platforms.

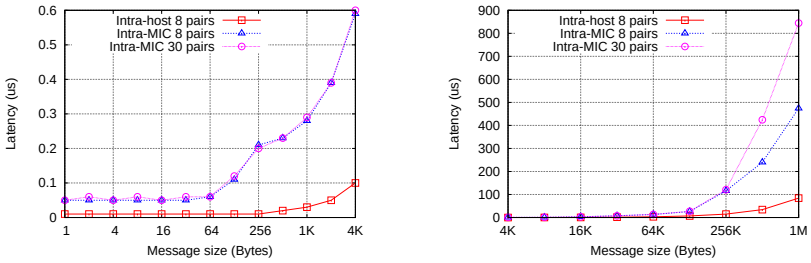
In Figure 4, we run the same benchmark with 16 threads on the host and with either 16 or 60 threads scattered on the MIC. For the MIC case, we have performed experiments with all 240 threads being used but as the performance was significantly worse, we have avoided including corresponding results. By using 16 threads on host and 60 threads on MIC, all of the cores on the CPU and coprocessors are fully occupied by UPC threads. For small messages, the memput latency for 256 byte message size with eight pairs of threads running on host and MIC are 0.01 and 0.2 μs, respectively. The memput latency for 256 byte message size with 30 pairs of threads running on MIC is 0.2 μs. We are able to observe that with increasing concurrent

small message communication (eight pairs to 30 pairs), MIC is able to keep the same latency up to 256 KB message size. This trend can be explained by the high bandwidth ring available on the MIC [2]. For large message size, the memput latency for 1 MB message size with 8 pairs of threads running on host or MIC are 100 and 480 μs, respectively. The memput latency of 1 MB message size with 30 pairs of threads running on MIC is 850 μs. We can observe that more concurrent large message transfers have more memory copy overhead, due to the limitation of the interconnects inside the MIC node.



(a) small message (b) large message

Figure 3: Intra-MIC Evaluations: single pair memput latency

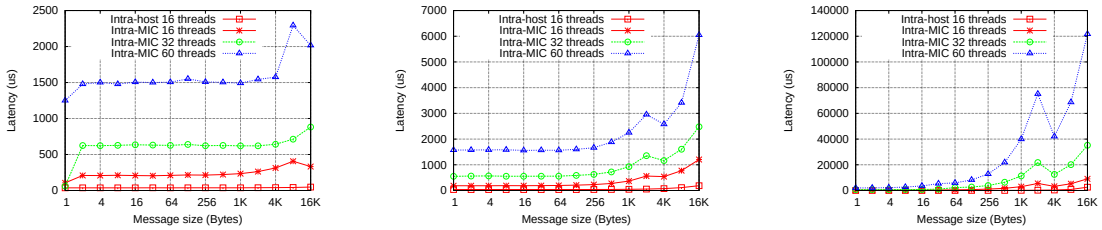


(a) small message (b) large message

Figure 4: Intra-MIC Evaluations: multi-pair memput latency

5.2.2 Native Mode Collective Evaluations

In this subsection, we evaluate the performance of UPC collectives running on host or MIC with a varying number of threads. We chose Bcast, Gather, and Exchange as representatives for common patterns across HPC applications. In Figure 5(a), we present Bcast results with 16 threads on host, 16, 32, and 60 threads on MIC, respectively. We could see that there are performance differences between running 16 threads on host and MIC. When we increase the number of threads for collective operations, the performance of collective operations are affected by increased memory contention. In Figures 5(b) and 5(c), we run the same set of experiments for Gather and Exchange operations. We could see that Gather and Exchange operations have the same performance trends as Bcast operation.

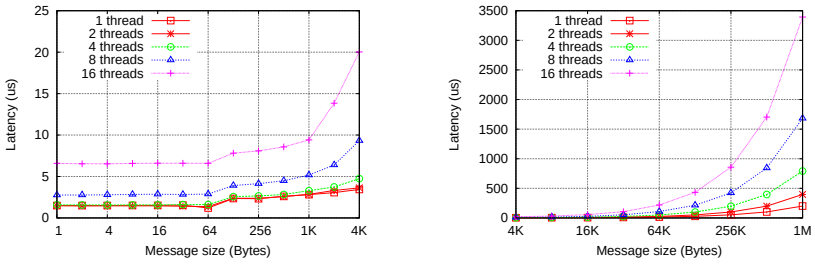


(a) Bcast (b) Gather (c) Exchange

Figure 5: Intra-MIC Evaluations: collective latency

5.2.3 Symmetric Mode Point-to-Point Evaluations

In this subsection, we evaluate the performance of memput operations across host and MIC in *symmetric* mode. We measure the performance from both host side and MIC side and show the results. In Figure 6, UPC threads launched on host CPUs initialize the `upc_memput` function calls while UPC threads on MIC are idle. We increase the number of threads involved in communication from 1 to 16. On the other hand, UPC threads on MIC initialize the `upc_memput` functions in Figure 7 from 1 thread to 60 threads. When the number of threads on the MIC and the host increases, the memput latency increases as a result of memory contention. While a single thread memput costs 1.5 μs and 16 concurrent memput costs 6.6 μs from host to MIC, as shown in Figure 6(a), MIC to host transfers perform significantly worse than host to MIC transfers as this involves the IB HCA reading from the MIC and this is known to suffer from the PCIe read bottleneck. In Figure 7(a), we observe that from MIC to host, memput costs 3.8 μs for single thread, 21 μs for 16 threads, and 51 μs for 60 threads. This indicates some of the bottlenecks involved in transferring data over the PCIe as traffic increases. In sum, the host-to-MIC and MIC-to-host performance is not symmetric in the *symmetric* mode.

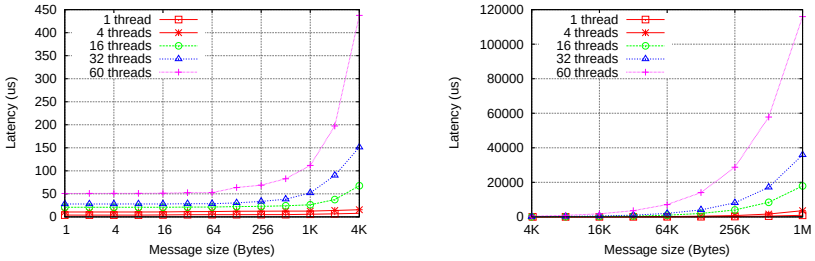


(a) small message (b) large message

Figure 6: Host-to-MIC Evaluations: multi-pair memput latency

5.3 Random Access

The Random Access benchmark is motivated by a growing gap in performance between processor operations and random memory accesses. This benchmark measures the peak capacity of the memory subsystem while performing random updates to the system memory. We carry out



(a) small message (b) large message

Figure 7: MIC-to-Host Evaluations: multi-pair memput latency

Random Access benchmark for *native* mode on host and MIC with full utilization. With table size equal to 2,097,152 words, the host *native* mode gets 93 μs while MIC *native* mode is 246 μs (Figure 8). We further evaluated RandomAccess benchmark in symmetric mode with total 76 UPC threads where 16 UPC threads on host and 60 UPC threads on MIC. The execution time dramatically increased from less than 1 second in *native* mode (both host and MIC) to 52 seconds in *symmetric* mode.

5.4 UTS Benchmark

The Unbalanced Tree Search (UTS) benchmark is a parallel benchmarking code that reports the performance achieved when performing an exhaustive search on an unbalanced tree [21]. In the benchmark, `upc_lock` is frequently utilized for load balancing. Fetching request is implemented by `upc_memget` operation. We evaluate the *native* mode on host and MIC and show the results in Figure 8. Due to the frequent small message exchange introduced by `upc_lock` and `upc_memget`, we can observe a 4X execution time on MIC of the execution time on host.

5.5 NAS Benchmark

In this section, we compare NAS benchmark performance with *native* mode and *symmetric* mode. In Figure 9, we present the results of NAS Class B performance. In both host and MIC *native* modes, the CPUs and coprocessors are fully occupied with 16 and 60 UPC threads, respectively. We are able to observe that 60 MIC coprocessors deliver 80%, 67%, and 54% performance as 16-CPU host for MG, EP, and FT. However for communication-intensive benchmarks CG and IS, the MIC spends 7x and 3x execution time, due to the bottleneck of slower intra-MIC communication, as we observed in Section 5.2.1.

In order to understand the scalability of host and MIC accordingly, we also measure the results of the same set of benchmarks with only one UPC thread on a CPU or a MIC core, respectively. We compare the Mop/s (Million operations per second) for each CPU or MIC core in the situations of single-thread and fully-occupied. As shown in Table 1, it is interesting to find out that, though MIC gives worse performance for the total execution time, it has less perforamnce degradation or better improvement for most of the benchmarks (except for IS) when the scale of the benchmarks increases. This evaluation indicates that MIC system could have a better scalability than the host system.

We also compare the FT Class B performance between *native* mode and *symmetric* mode. Four threads with *native* mode on host takes 22 seconds for FT Class B test. However, when we

	CG		EP		FT		IS		MG	
	host	MIC	host	MIC	host	MIC	host	MIC	host	MIC
single-thread	536	12	23	4	1248	103	85	12	2930	249
fully-occupied	602	18	23	4	580	80	57	4	854	189
ratio	12.3%	52.5%	0%	0%	-53.5%	-21.7%	-33%	-65%	-70%	-23%

Table 1: NAS benchmarks with native mode: Mop/s per core

launch the job through *symmetric* mode with four UPC threads on host and MIC, respectively, the total execution time for FT benchmark is 116 seconds.

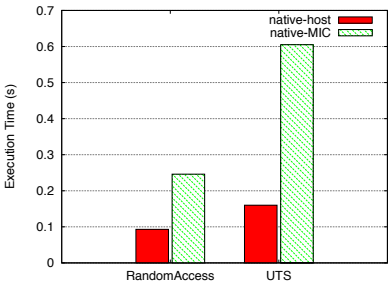


Figure 8: *RandomAccess and UTS benchmark performance with native mode*

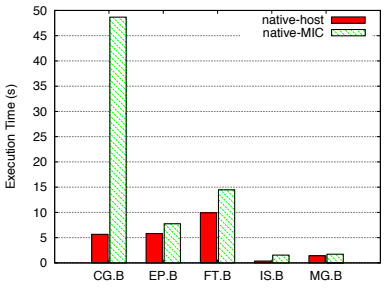


Figure 9: *NAS benchmark performance with native mode (Class B)*

During the evaluation of NAS benchmark, we also experienced unbalanced memory problem for the *symmetric* mode. As shown in Figure 1, MIC system has much less local memory (8G) than host node (32G). In *symmetric* mode, the size of shared array region on host must match the size of shared array region on MIC. When the problem size of NAS benchmark increases, the MIC memory can be easily used up while the host memory is still available. One of the benefits of *symmetric* and *native* modes is that UPC applications can run on MIC with no change to source code. This means coprocessors are assigned with the same workload as CPUs. However, considering the unbalanced physical memory size and computation power, optimizations to assign corresponding workload and shared memory region on coprocessors and CPUs would be an important consideration in this direction.

6 Related Work

There have been many researches to utilize MIC to accelerate parallel applications. Being a new terrain, optimal programming model for writing parallel applications on MIC is yet to be explored. In [18], Sreeram et. al presented their experience with MVAPICH2 MPI library on MIC architecture. They have tuned and modified the MVAPICH2 library to provide better performance for basic point-to-point and collective operations running on MIC architecture. Article [19] proposes designs that are based on standard communication channels and Intel's low-level communication API (SCIF) to optimize MPI communications on clusters with Xeon Phi coprocessors. They are able to show performance improvements of running applications on

MIC. Besides existing works in MPI, there are also some works using UPC (Unified Parallel C) on MIC. Article [20] describes SGI compiler support for running UPC application on MIC.

Global Address Space Programming Interface (GASPI) [10] is a PGAS API. GASPI allows software developer to map memory of Xeon Phi to RDMA PGAS memory segments. Those segments could be directly read/write from/to between processes – inside nodes and across nodes. Those RDMA operations could significantly minimize synchronization overheads. In this paper, we explore the performance benefits of using UPC on MIC. We provide a detailed evaluation of UPC-level benchmarks and applications.

7 Conclusion

In this paper, we discussed the intra-node memory access problems and host-to-MIC connection issues for running UPC applications on a MIC system under *native* and *symmetric* programming modes. We chose the multi-threaded UPC runtime with the new proposed "leader-to-all" connection mode according to the discussion. We evaluated point-to-point, collectives, Random Access test and NAS/UTS benchmarks for *native* mode. We also examined the point-to-point communication performance between host and MIC in *symmetric* mode. According to the evaluation results, we found out several significant problems for UPC running on many-core system like MIC, such as the communication bottleneck between MIC and host, and the unbalanced physical memory and computation power issues. For the future research directions, deploying delegate to handle communications between MIC and host in a contention-aware manner could help alleviate the high asymmetric memory access overhead. Optimizations to assign corresponding workload and shared memory region across coprocessors and host CPUs would be an important consideration for *symmetric* mode.

References

[1] Intel Manycore Platform Software Stack. http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss.

[2] Intel MIC Architecture. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[3] Symmetric Communication Interface. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-software-developers-guide.pdf.

[4] TOP 500 Supercomputer Sites. http://www.top500.org.

[5] F. Blagojevic, P. H. Hargrove, C. Iancu, and K. A. Yelick. Hybrid PGAS Runtime Support for Multicore Nodes. In *Fourth Conference on Partitioned Global Address Space Programming Model*, 2010.

[6] M. Deisher, M. Smelyanskiy, B. Nickerson, V. W. Lee, M. Chuvelev, and P. Dubey. Designing and Dynamically Load Balancing Hybrid LU for Multi/Many-core. *Comput. Sci.*, 26(3-4):211–220, June 2011.

[7] J. Dongarra. Visit to the National University for Defense Technology Changsha, China. http://www.netlib.org/utk/people/JackDongarra/PAPERS /tianhe-2-dongarra-report.pdf.

[8] J. Duell. Pthreads or Processes: Which is Better for Implementing Global Address Space Languages? Master's thesis, University of California at Berkeley, 2007.

[9] Editor: Dan Bonachea. GASNet specification v1.1. Technical Report UCB/CSD-02-1207, U. C. Berkeley, 2008.

[10] Global Address Space Programming Interface. http://www.gaspi.de/.

[11] K. Hamidouche, S. Potluri, H. Subramoni, K. Kandalla, and D. K. Panda. MIC-RO: Enabling Efficient Remote Offload on Heterogeneous Many Integrated Core (MIC) Clusters with InfiniBand. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 399–408, New York, NY, USA, 2013.

[12] IBM. Message Passing Interface and Low-Level Application Programming Interface (LAPI). http://www-03.ibm.com/systems/software/parallel/index.html.

[13] InfiniBand Trade Association. http://www.infinibandta.com.

[14] J. Jose, M. Luo, S. Sur, and D. K. Panda. Unifying UPC and MPI Runtimes: Experience with MVAPICH. In *Fourth Conference on Partitioned Global Address Space Programming Model (PGAS)*, Oct 2010.

[15] Lawrence Berkeley National Laboratory and University of California at Berkeley. Berkeley UPC - Unified Parallel C. http://upc.lbl.gov/.

[16] M. Luo, J. Jose, S. Sur, and D. K. Panda. Multi-threaded UPC Runtime with Network Endpoints: Design Alternatives and Evaluation on Multi-core Architectures. In *Int'l Conference on High Performance Computing (HiPC '11)*, Dec. 2011.

[17] Open Fabrics Workshop 13. OFS Software for the Intel Xeon Phi TM. https://www.openfabrics.org/ofa-documents/presentations/doc_download/539-ofs-for-the-intel-xeon-phi.html.

[18] S Potluri, K Tomko, D Bureddy, and D. K. Panda. Intra-MIC MPI Communication using MVAPICH2: Early Experience. In *TACC-Intel Highly-Parallel Computing Symposium*, 2012.

[19] S. Potluri, A. Venkatesh, D. Bureddy, K. Kandalla, and D. K. Panda. Efficient Intra-node Communication on Intel-MIC Clusters. In *13th IEEE Int'l Symposium on Cluster Computing and the Grid (CCGrid13)*, 2013.

[20] SGI. Unified Parallel C (UPC) User Guide. http://techpubs.sgi.com/library/manuals/5000/007-5604-005/pdf/007-5604-005.pdf.

[21] The Unbalanced Tree Search Benchmark. http://sourceforge.net/p/uts-benchmark/wiki/Home/.

[22] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

Hot Papers

These papers were accepted into the hot category and are works demonstrating results, ideas, applications and models which are still under active development.

Asynchronous Global Heap: Stepping Stone to Global Memory Management

Yuichiro Ajima[12], Hideyuki Akimoto[12], Tomoya Adachi[12], Takayuki Okamoto[12], Kazushige Saga[12], Kenichi Miura[12] and Shinji Sumimoto[12]

[1] Fujitsu Limited, Kawasaki, Kanagawa, Japan
[2] JST CREST, Kawaguchi, Saitama, Japan
{aji, h.akimoto, adachi.tomoya, tokamoto, saga.kazushige, k.miura, sumimoto.shinji}@jp.fujitsu.com

Abstract

Memory utilization for communication will become a major problem in the exascale era because an exascale parallel computation job will include millions of processes and a conventional communication layer requires preprovisioned memory of a size proportional to the number of processes. One approach to addressing the memory utilization problem is to allocate a data sink on a remote node dynamically each time a node needs to send data to the remote node. In this paper, a memory management scheme that can provide memory to a process in a remote node is called "global memory management." A global memory management scheme that can be accessed via interconnect is an ideal solution that does not waste local processing resources but is also different from today's local memory management schemes. For a stepping stone to global memory management, we propose an asynchronous global heap that virtually achieves global memory management while minimizing modifications to the operating system and the runtime. In addition, new hardware features for global memory management are also co-designed.

1 Introduction

Cluster-type parallel computers are the mainstream of today's high performance computing (HPC). A cluster is composed of a massive number of stand-alone computers that are interconnected. On a stand-alone computer, a network interface is an expansion device and not a first-class citizen such as a processor, memory, and operating system. An ordinary network interface incurs ten or more microseconds of latency [6] because of a data delivery scheme that involves context switches. In contrast, today's HPC interconnect devices access remote memory in the sub-microsecond range [3, 4, 1] at the price of overheads establishing a connection to the remote node and registering memory to the device. These overheads are permitted as a tradeoff to resolve performance bottlenecks. Therefore, today's interconnect designs target large or iterative data transfers.

Memory utilization for communication will become a major problem in the exascale era because an exascale parallel computation job will include millions of processes and conventional communication layers require preprovisioned memory of a size proportional to the number of processes in order to avoid performance bottlenecks. One approach to addressing the memory utilization problem is to allocate a data sink on a remote node dynamically each time a node needs to send data to the remote node. In this paper, a memory management scheme that can provide memory to a process in a remote node is called "global memory management," and memory provided by the global memory management scheme is called "global memory." A global memory management scheme that can be accessed via interconnect is an ideal solution that does not waste local processing resources but is also different from today's local memory management schemes.

For a stepping stone to global memory management, we propose an asynchronous global heap that virtually achieves global memory management while minimizing modifications to the operating system and the runtime. In addition, new hardware features for global memory management are also co-designed assuming a system-on-chip whose device I/O bus is customized to enhance the memory access capability of the integrated interconnect device.

In this paper, we propose an asynchronous global heap as a primitive global memory management scheme. In Section 2, the asynchronous global heap is introduced. In Section 3, the results of evaluating the asynchronous global heap functions are shown. In Section 4, this work is summarized, and possible future works are listed.

2 Asynchronous Global Heap

We propose an asynchronous global heap that includes a concurrent data structure allocated on each process and application programming interfaces (API) to provide free memory to other processes. Control variables and a heap body of the data structure are designed to be accessed via the remote direct memory access (RDMA) features of an interconnect device. Therefore, the control variables and heap body are allocated in a memory region that can be accessed via RDMA. With today's operating system and runtime, RDMA capable memory regions are required to be pinned and registered to an interconnect device prior to RDMA access. Therefore, the data structure is placed in a locally provided memory region, and the free memory managed by the operation system decreases even though the heap body is totally unused at the time. We choose memory allocators of runtimes including the malloc() function to resolve this prior memory consumption problem while leaving the operating system and the memory registration scheme unchanged. The implementation of an asynchronous global heap uses only memory allocation system calls of the operating system, so the memory allocator of a runtime can obtain free memory from an asynchronous global heap.

2.1 Data Structure

Figure 1 shows the data structure of an asynchronous global heap. To isolate fragmentation caused by local and remote memory allocation, local memory allocation consumes free memory from the top, and global memory allocation consumes free memory from the bottom. A control variable that indicates the bottom of the free memory is called a "global break" (gbrk), and another control variable that indicates the bottom of the memory allocated locally is called a "global limit" (glimit). The gbrk variable may be changed by global memory allocators, and the glimit variable may be changed by a local memory allocator. All memory allocators using the asynchronous global heap read both the gbrk and glimit variables to calculate the size of the free memory space, so accessing these variables requires exclusive control.

Each process holds a dynamic state of its own asynchronous global heap that consists of three control variables: a lock, the gbrk, and the glimit. A control structure including these control variables is placed in conjunction with the heap body. Caching the dynamic state of another process is prohibited.

2.2 Application Programming Interfaces

The API design of an asynchronous global heap is similar to the data segment system calls of Linux which includes the brk() system call. There are five API functions. The ginit() function initializes the asynchronous global heap. The gbrk() function changes the global break.

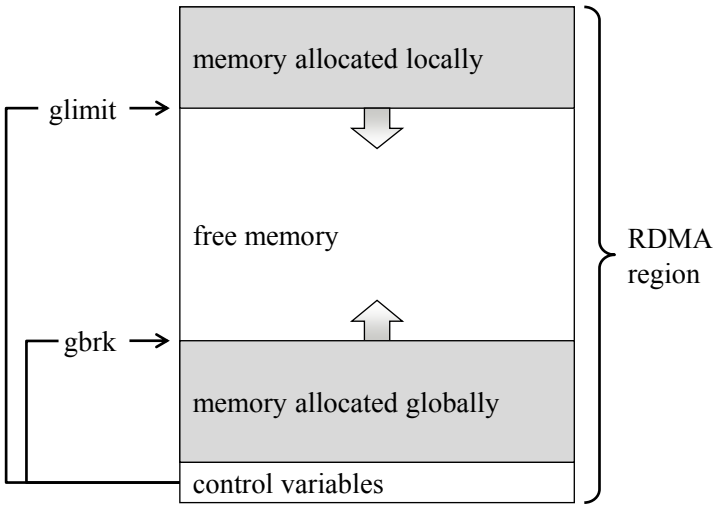


Figure 1: Data structure of asynchronous global heap

The sgbrk() function moves the global break. The gglimit() function changes returns the free memory information. The sglimit() function changes the global limit. To obtain free memory from other nodes, gbrk(), sgbrk(), and gglimit() functions have an input argument to specify a process identification number. Initializing an asynchronous global heap involves a collective communication process that gathers the identifiers of RDMA capable memory regions of whole processes. API functions other than the ginit() do not register local memory or exchange identifiers of registered memory.

2.3 Co-designing New RDMA Features

Global memory obtained from an asynchronous global heap can be accessed with ordinary RDMA put and get features. To access the control variables effectively, we introduce three additional RDMA features: RDMA atomic compare and swap (CAS), RDMA remote fence, and interoperable atomic operations. RDMA atomic CAS sequentially operates compare and swap on remote memory without interruption from any other RDMA accesses and effectively performs mutual exclusion of the control structure. RDMA remote fence forces memory accesses of the RDMA requests sent prior to the fence completed before the memory accesses of RDMA requests sent after the fence are started. Implementing a RDMA remote fence feature that handles memory access ordering remote-side may reduce the latency of reading control variables because an RDMA get request can be sent speculatively immediately after the RDMA remote fence following the RDMA atomic CAS, which tries to acquire the lock. Interoperable atomic operations are carefully implemented atomic operations of the processor atomic instructions and RDMA atomic operations, so processor atomic instructions are ensured not to be interrupted by any RDMA memory accesses, and RDMA Atomic operations are ensured not to be interrupted by any processor memory accesses. Interoperable atomic operations allow any control variable of an asynchronous global heap to be accessed by processor memory access instructions as long as the control variables belong to the accessing process itself.

3 Evaluation

3.1 Evaluation Environment

In this chapter, we evaluate the execution time of the gbrk(), gglimit(), and sglimit() functions that are assumed to be called frequently from memory allocators that support asynchronous global heap. A prototype system of the K computer placed in Fujitsu's Numazu Plant was used for the evaluations. The processor was a SPARC64^{TM} VIIIfx [5], and it has an operating frequency of 2 GHz and eight cores. The interconnect device was a Tofu interconnect [2]. The experiment programs used the Message Passing Interface (MPI) and were executed with two MPI processes. The MPI process rank 0 repeatedly executed an asynchronous global heap API function 1001 times to access the same asynchronous global heap. The experiment programs measured the time between starting the second execution of the function and finishing the last one.

The experiment programs used the Tofu library (tlib) in conjunction with MPI for RDMA communication. The tlib is a low-level API for using features of the Tofu interconnect directly. Each MPI process created a thread to emulate RDMA atomic CAS and interoperable atomic operations. The RDMA remote fence was implemented by using the strong order flag feature of the Tofu interconnect [1].

The average execution time of each function was measured for each of the four different combinations of access patterns and assumed RDMA features. For the remote access patterns, the MPI process rank 0 accessed its own asynchronous global heap, and there were two options: assume or do not assume the RDMA remote fence feature. For the local access patterns, the MPI process rank 0 accessed an asynchronous global heap on MPI process rank 1, and there were two options: assume or do not assume the interoperable atomic operations feature.

3.2 Evaluation Result

Figure 2 shows the evaluation results. The graph shows the measured average execution time of the asynchronous global heap API functions. The vertical axis has a logarithmic scale, and the unit of time is microseconds. The results of local access with RDMA atomic were shorter than those of remote access by 35 to 48%. The difference comes from the access method for controlling variables other than the lock variable. For the local access patterns, only the lock variable was accessed by RDMA, and the others were accessed by processor instructions. For the remote access patterns, all control variables were accessed by RDMA. The results of remote access with fence were shorter than those for remote access without fence by 25 to 37%. The difference comes from speculative accesses to the control variables with fence that hide the latencies of mutual exclusion. As for hiding mutual exclusion latencies, the results of remote access with fence were closer to those of local access with RDMA atomic rather than those of remote access without fence. For the local access patterns, interoperable atomic operations reduced the execution time by 99% because all control variables were accessed by processor instructions when interoperable atomic operations were assumed.

4 Summary and Future Work

In this paper, we proposed an asynchronous global heap as a stepping stone for an ideal global memory management scheme that allows other nodes to obtain memory directly via an interconnect. An asynchronous global heap provides free memory to both local and global memory

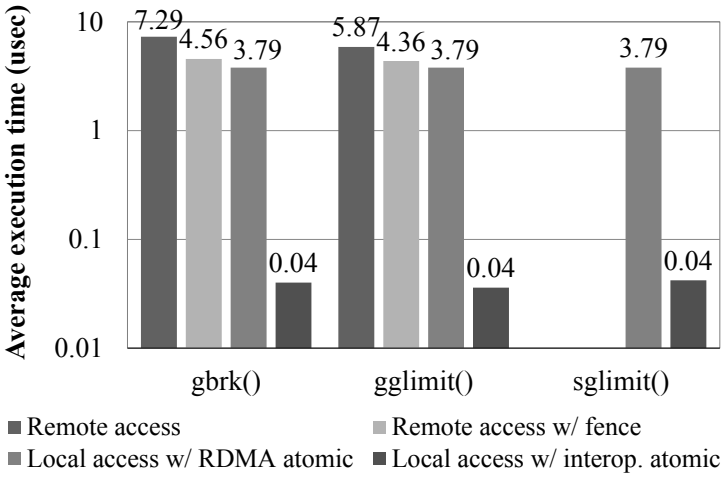


Figure 2: Evaluated average execution time of asynchronous global heap API functions

allocators. For efficient access to control variables, three RDMA features were also introduced: RDMA atomic CAS, RDMA remote fence, and interoperable atomic operations. The results of evaluating the API functions showed that RDMA remote fence reduced access time to an remote asynchronous global heap by 25 to 37%, and interoperable atomic operations reduced access time to a local asynchronous global heap by 99%.

Our possible future work includes investigating an extensible asynchronous global heap, another global memory management scheme that causes no fragmentation, low-latency algorithms for global memory allocators to merge fragments on deallocation, and intelligent strategies for user programs to reuse allocated data sinks.

References

[1] Yuichiro Ajima, Tomohiro Inoue, Shinya Hiramoto, Toshiyuki Shimizu, and Yuzo Takagi. The Tofu Interconnect. *IEEE Micro*, 32(1):21–31, 2012.

[2] Yuichiro Ajima, Shinji Sumimoto, and Toshiyuki Shimizu. Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers. *IEEE Computer*, 42(11):36–40, 2009.

[3] Robert Alverson, Duncan Roweth, and Larry Kaplan. The Gemini System Interconnect. In *Proceedings of IEEE 18th Annual Symposium on High Performance Interconnects*, pages 83–87, 2010.

[4] Dong Chen et al. The IBM Blue Gene/Q Interconnection Network and Message Unit. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, number 26, 2011.

[5] Takumi Maruyama et al. SPARC64 VIIIfx: A New-Generation Octocore Processor for Petascale Computing. *IEEE Micro*, 30(2):30–40, 2010.

[6] Justin (Gus) Hurwitz and Wu-chun Feng. End-to-end performance of 10-gigabit ethernet on commodity systems. *IEEE Micro*, 24(1):10–22, 2004.

UPCGAP: A UPC package for the GAP system

Nick Johnson[1], Alexander Konovalov[2], Vladimir Janjic[2] and Steve Linton[2]

[1] EPCC, The School of Physics and Astronomy, University of Edinburgh
Nick.Johnson@ed.ac.uk
[2] School of Computer Science, University of St Andrews

Introduction

This paper presents current work in adding Unified Parallel C (UPC) support to the GAP computer algebra system [3]. Our work is motivated by the need to parallelise *orbit enumeration*, a central concept for many areas of discrete mathematics. This algorithm requires the storage of, and access to, potentially huge numbers of objects. UPC's memory model allows large, distributed data structures whose memory requirements exceed the amount of memory available on any single node. It can make use of the memory available to multiple nodes in a cluster or HPC system, while still using the normal C-array syntax. We describe the main functionality of UPC-extended GAP, show and discuss some preliminary performance results and consider further improvements.

This paper is structured as follows: in Section 1 we provide an overview of the GAP system; in Section 2 we describe the orbit enumeration problem and consider parallelisation strategies; in Section 3 we introduce the implemetation of UPCGAP; Sections 4 and 5 discuss results and provide pointers to future work respectively.

1 The GAP system

The GAP system [3] is an open-source system for discrete computational algebra. It consists of a kernel written in C which provides the run-time environment, memory management and an interpreter for the GAP language. It also contains a library written in the GAP language and implementations of various mathematical algorithms, along with data libraries such as the library of groups of small orders and the character table library. GAP has been developed since 1986 and has a substantial user base, being used for research and teaching at hundreds of institutions worldwide.

The GAP programming language uses a very sophisticated type system which supports dynamic polymorphism and is fully described in [2]. Currently the language is being extended with primitives and higher-level parallel skeletons for shared and distributed memory programming as part of the HPC-GAP project.

Adding UPC support to GAP comes with many challenges. Firstly, functionality to work with UPC constructs should be available at the GAP language level. Secondly, the GAP kernel requires adaptation to be compiled with a UPC compiler, and new kernel modules require adherence to GAP kernel programming rules. Most importantly, GAP operates with objects having more complex structure than floats or integers: managing these in UPC memory constructs is a non-trivial task. Finally, to be useful and easily used by the GAP community, the most suitable form of organising the UPC extension is a GAP package.

2 Parallel orbit enumeration

Orbit enumerations represent an important class of algorithms that have many applications in computational discrete mathematics, e.g. determinisation of finite-state automata, parsing formal languages and group theory. The orbit enumeration problem can be formally described in the following way:

Definition 1 (Orbit of an element)**.** *Let X be a set of points, G a set of generators, $f : X \times G \to X$ a function (where $f(x, g)$ will be denoted by $x \cdot g$) and $x_0 \in X$ a point in X. The* orbit *of x_0, denoted by $\mathcal{O}_G(x_0)$, is the smallest set $M \subseteq X$ such that:*

1. *$x_0 \in M$*
2. *for all $y \in M$ and all $g \in G$ we have $y \cdot g \in M$.*

The basic sequential orbit enumeration algorithm starts with two lists: $M = [x_0]$, the orbit points discovered so far and $U = [x_0]$, a set of unprocessed points. In each step, we remove the first point x from U, apply all generators $g \in G$ to x and add the resulting points $x \cdot g$ that are not in M to M and U. This terminates when all $g \in G$ have been applied to all $x \in M$, discovering the whole of the orbit $\mathcal{O}_G(x_0)$. The number of entries in M gives the length of the orbit. Pseudo-code for solving this can be seen in Algorithm 1.

The key observation that makes the parallelisation of this algorithm possible is that each generator $g \in G$ can be applied to each point from U *in parallel*. However, it becomes apparent that parallel threads will have to synchronise their access to both U and M. A common structure for M is a hash table and for U is a queue, both of which are implemented as part of our package. It is possible to allow parallel access to different parts of M by different parallel threads with the caveat that only one thread may access one location at any time. At first, a suitable parallelisation strategy seems to use one thread per generator $g \in G$. This way, multiple generators would be applied to each point in parallel. However, in many practical problems the size of G is small, restricting parallelization.

Algorithm 1 Sequential Orbit enumeration algorithm

```
M ← x_0
U ← x_0
while U ≠ ∅ do
    x ← element from U
    for g ∈ G do
        y ← x · g
        if y ∉ M then
            M ← y
            U ← y
        end if
    end for
end while
```

Therefore, we propose the strategy where each parallel thread executes the following loop:

1. Test whether the set of unprocessed points U is empty;
2. If U is not empty, remove from U a set of points X such that $|X| \leq N$, and apply each generator $g \in G$ to each point from X, producing a set of points Y;

3. Add to M and U all points from Y that are not already in M;

4. If U is empty, wait until it becomes non-empty, or until the termination condition is met.

Pseudo-code for this can be seen in Algorithm 2. *exit_test* is function which returns 1 if all threads have no work to do, and 0 otherwise.

Algorithm 2 Parallel Orbit enumeration algorithm

```
M ← x_0
U ← x_0
if f ≠ 1 then
    while U ≠ ∅ do
        X ← min(N, |U|) elements from U
        for g ∈ G do
            for x ∈ X do
                y ← x · g
                if y ∉ M then
                    M ← y
                    Y ← y
                end if
            end for
        end for
        U ← Y
    end while
    f ← exit_test
else
    Exit
end if
```

3 A UPC kernel for GAP

UPC [1], a PGAS language extension to C, is implemented by both open source (Berkeley UPC, GUPC) and commercially available (Cray) compilers. Its memory model allows access to any element of an array stored in the *shared* memory space by any parallel thread using standard array syntax. UPC makes it possible to have efficient distributed data structures whose memory requirements exceed the amount of memory available on any single node. Since orbit enumeration often requires storage of a large number of objects, this explains our motivation to extend GAP with UPC support.

The UPCGAP package adds to the GAP kernel two UPC data structures, namely a distributed hash-table and a queue, as well as functions to manipulate these using the GAP language. Both the hash-table and the queue must be able to store a number objects of a fixed size; these are determined by the problem to be solved. Dynamic memory allocation is used for both structures allowing efficient use of memory at the cost of potential efficiency gains from blocked memory layout.

The hash-table structure comprises two parts: one for the storage of objects (the object table), and one which indicates if a corresponding location in the object table is empty or full (the

indicator table). We seek, at this stage, only to enumerate the objects and therefore provide a function which both tests for membership and stores objects should they not exist in the object table. The object and indicator tables are allocated in UPCs default manner; if location x is stored in the shared memory space affine to thread t, location $x+1$ is stored in the space affine to thread $t+1$. As dynamic memory allocation is used, the blocking factor is not preserved and we use a simple algorithm to determine on which thread each location resides. Hash clashes (where two different objects hash to the same location) are handled by searching the indicator table to find the next empty location. By using an array of locks (one per thread) to control write access to the hash-table, we are able to allow parallel access albeit in a coarse grained manner.

The queue structure likewise comprises two parts: one for the storage of objects and a pointer which indicates the location holding the head of the queue. We provide two access functions for the queue: one to add and another to remove an object. A single lock prevents concurrent manipulation of the queue by more than one thread.

4 Results

To gain an understanding of the performance of our parallel implementation of orbit enumeration, we use an example from group theory: the sporadic finite simple Harada-Norton (HN) group in its 760-dimensional representation over the field with 2 elements, acting on vectors. The set X of all possible points consists of about 10^{228} elements. The size of the orbit used in our test is 1140000 points. We compute this orbit using four different chunk-sizes: 10, 100, 1000 and 10000. The chunk-size is the maximum number of elements which may be removed from the queue by a thread in one iteration, denoted by N in Algorithm 2.

The code was run both on HECToR, the UK national supercomputing service, using the Cray UPC compiler, and also on Ladybank, an 8-core shared memory machine based at St Andrews, with Intel Xeon X5570 2.93 GHz processors and 64 GB RAM, using GUPC compiler.

Figure 1 shows the speedups of our parallel implementation over sequential orbit enumeration using 1–32 threads on HECToR. We observe that when fewer cores (≤ 8) are used we obtain sublinear, but reasonable, speedup, and that the speedup is mostly independent of the chunk size. When more cores are used, we can observe a drop in performance for all chunk sizes, except the largest one. We suspect that the performance drop is due to contention in accessing shared resources (hash-table and queue). The larger the chunk size is, the less frequently parallel threads access these shared data structures, and this results in a better performance.

Figure 2 shows how the size of the task queue changes over time during the orbit calculation for different chunk-sizes at a fixed thread count on Ladybank. This illustrates the non-regular nature of the orbit calculation. We can observe that for smaller chunk sizes, the size of the task queue is large and grows constantly until the final stage. This indicates that threads are not able to take advantage of all available parallelism and is most likely due to them spending time waiting for access to shared data structures. As the chunk-size increases, we can observe that queue sizes are smaller and they do not increase dramatically over time. This is likely due to threads being able to do more work as they need less frequent access to shared data structures.

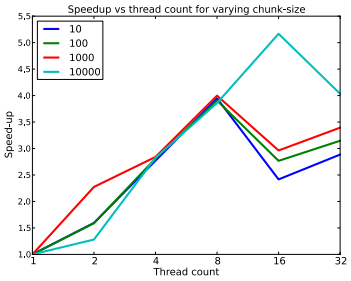


Figure 1: Performance for enumerating the orbit of the HN group for varying thread counts and chunk-sizes.

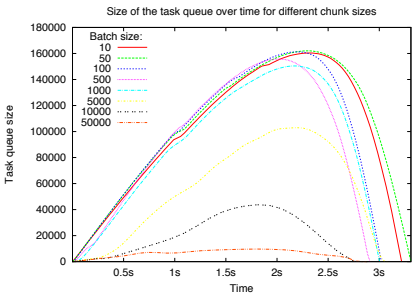


Figure 2: Irregularity of orbit calculation on 8 cores using varying chunk sizes.

5 Conclusions and Future Work

In this paper we have presented UPCGAP, a package for extending the GAP system to make use of the large memories and parallel processing provided by UPC. We have shown that with the addition of some simple data structures and manipulation functions, we are able to provide some modest scaling in our example application.

We suggest that additional scaling and performance may come from optimisation of the hash table and queue implementations. In particular future work will focus on increasing the granularity of the locks of the hash-table. It may also be possible to extend chunking to the storage of objects in the hash-table, which, combined with the use of non-blocking locks, could reduce contention. We also plan to run more experiments with different orbit instances on different parallel architectures.

6 References and Acknowledgements

The work is supported by the project "HPC-GAP: High Performance Computational Algebra and Discrete Mathematics" (EP/G055181/1, EP/G055742/1).

References

[1] UPC Language Specifications, v1.2. Technical Report, UPC Consortium, 2005.

[2] Thomas Breuer and Steve Linton. The GAP 4 type system: organising algebraic algorithms. In *Proceedings of the 1998 international symposium on Symbolic and algebraic computation*, ISSAC '98, pages 38–45, New York, NY, USA, 1998. ACM.

[3] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.6.5*, 2013.

First Experiences on Collective Operations with Fortran Coarrays on the Cray XC30

Pekka Manninen[1] and Harvey Richardson[2]

[1] Cray Finland manninen@cray.com
[2] Cray UK harveyr@cray.com

Abstract

The Cray XC30 is the latest Cray supercomputer architecture. It is a promising platform for the Partitioned Global Address Space programming model because of its all-to-all like, remote direct memory access capable Aries interconnect complemented by the Cray programming environment having the most advanced Fortran coarrays support so far. We evaluate the performance of some collective communication operations implemented with Fortran coarrays on the Cray XC30 architecture and compare them with blocking and non-blocking Message-Passing Interface collectives.

1 Introduction

The Partitioned Global Address Space (PGAS) programming model provides a global view of the memory across nodes and supports one-sided access to shared data. The new Cray XC30, or Cascade, architecture is a promising platform for utilizing PGAS languages due to its remote direct memory access (RDMA) supporting hardware and an all-to-all style interconnect topology as well as programming environment featuring compiler, runtime and tool support for several PGAS languages.

It is often necessary to have application phases involving a large set of processes working collectively to perform a global (across all processing elements) communication operation. These collective operations, such as data replication, task synchronization, and reduction can be straightforwardly performed in PGAS languages by accessing shared variables in a global address space. In two-sided communication models such as the message-passing interface (MPI) [1] they exist as standalone procedures. In subsequent sections we report some first experiences on how the collective communication patterns implemented with Fortran coarrays perform on the XC30. Even if the PGAS collectives are available in optimized libraries such as the GASNet library [2], we will study straightforward implementations to simulate cases where a programmer wants to integrate the collective operations in a coarrays code to enable overlapping computation and data addressing, as well as to have full control on synchronization. Moreover, by deliberately using compact, non-hand-tuned implementations, we are able to comment on how well the compiler recognizes and implements the communication patterns. We have implemented all MPI non-vector collective corresponds and will focus on two of them as an example. These implementations are available in the CRESTA Collective Communication Library [3]. Those who are interested in the implementation details can obtain the source code from the authors.

2 Observations

2.1 Example of a rooted collective: data replication

Although the coarrays syntax allows for a very elegant and simple `MPI_Bcast` style data transfer expressed on a single source line, there is a danger that all tasks accessing the memory of one

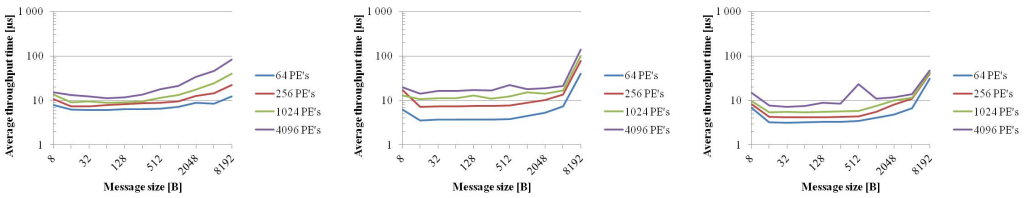


Figure 1: The average throughput of the data replication (Bcast) operation with coarrays (left), `MPI_Bcast` (middle) and `MPI_Ibcast` (right).

image simultaneously may become a bottleneck. In principle, the compiler could recognize and implement such an expression with a more efficient communication pattern.

We compared several approaches for data replication. From these, we observed that an approach where the data is first fetched from the root process by square root of num_images() images, and then the remaining get their data from one of these images, was the fastest one. Even this most elaborate approach (still consisting of 7 lines of code, however) is being outperformed by `MPI_Bcast` and especially `MPI_Ibcast` for small communicators and messages. With 1024 and 4096 images the coarray approach is marginally faster than `MPI_Bcast` for larger messages. However, the differences are not dramatic, and the coarray approach appears as a valid alternative for `MPI_Bcast` style communication patterns, especially since this kind of communication seldom becomes a bottleneck. The comparison of the more elaborate coarrays broadcast with `MPI_Bcast` and `MPI_Ibcast` are presented in Figure 2.1 as average times to complete the operation (i.e. a smaller value is faster).

2.2 Example of non-rooted collectives: all-to-all data exchange

The all-to-all data exchange pattern is a scalability bottleneck often encountered in supercomputing applications: because the number of messages increases as a square of the number of MPI tasks. It is also encountered in key algorithms in scientific computing, for example in Fast Fourier Transforms.

We implemented the Alltoall with coarrays using two different approaches: Directly looping over the full set of images, each image fetches its portions from the other; and another in which the fetching loop is split over images, to a loop running from `this_image()+1` to `num_images()` and to a subsequent loop running from 1 to `this_image()-1`. This is to avoid several simultaneous fetches from the same image.

Of these, the second implementation was faster. The comparison of the former implementation with `MPI_Alltoall` and `MPI_Ialltoall` are presented in Figure 2.2. These are again average times needed to complete the operation. Coarrays are again a viable alternative outperforming both `MPI_Alltoall` and `MPI_Ialltoall` starting from somewhat larger message sizes.

Of other all-to-all collectives, a coarray implementation of the reduce-scatter operation utilizing the upcoming coarray intrinsics (here `co_sum`) was able to outperform the MPI version. For the all-reduce (`MPI_Allreduce`) or gather-to-all (`MPI_Allgather`) operations, no coarray implementation was established that would have outperformed the MPI version.

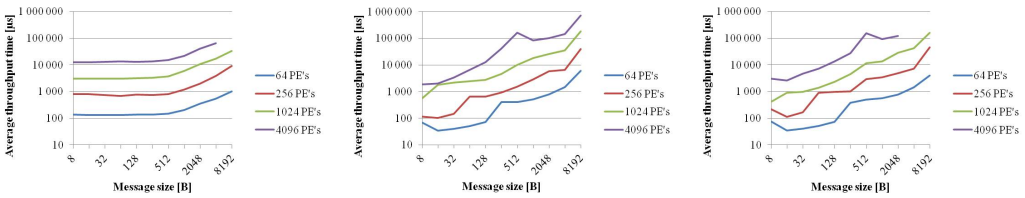


Figure 2: Performance of all-to-all data exchange with coarrays (left) and with blocking (middle) and non-blocking MPI (right).

3 Concluding remarks

The Cray XC30 architecture is a very capable PGAS programming platform; benefitted by the RDMA-capable, all-to-all style, interconnect together with the programming environment support including an optimizing compiler and a performance analysis suite CrayPAT that natively supports PGAS languages.

We conclude that outperforming MPI in collective operations with the coarrays approach is challenging, but the coarray programming model is an alternative for MPI even when employing collective communication patterns. The straightforward and compact coarray alternatives were sometimes faster than MPI, especially in the intensive data exchange as available in the `MPI_Alltoall` procedure, at least up to 4096 MPI tasks / coarray images. The performance difference depends on the communicated data size and the number of tasks / images, coarrays being faster starting from messages of a few hundred bytes. We found also that the coarrays implementation of the reduce-scatter operation (cf. `MPI_Reduce_scatter`) was able to outperform the MPI implementation.

Since the MPI library of XC30 precedes to the libraries in the earlier XT and XE Cray product lines, i.e. is a product of years of intense development and optimization efforts, we tend to believe the cases where coarrays outperformed MPI are not due to non-optimality or immaturity of the MPI library but rather an example of capabilities of the PGAS approach in this architecture.

Our observations on best practice for coarray programming on the XC30 include the following:

- Implementing multiple intermediate roots in rooted (one-to-all or all-to-one) collectives is necessary to achieve performance that would be even comparable with MPI.
- Avoiding multiple simultaneous accesses to an image is important, but such load balancing can be achieved by quite simple approaches.

Whether it is possible to overlap computation and communication better with coarray collectives than with the non-blocking collectives of MPI 3.0, which are available on XC30 already, is left for a later study.

References

[1] Message Passing Interface Forum. Mpi: A message-passing interface standard version 3.0. Technical report, 2012.

[2] P. H. Hargrove K. A. Yelick R. Nishtala, Y. Zheng. *Parallel Computing*, (39):576–591, 2011.

[3] P. Manninen. Non-blocking collectives run-time library. Technical Report D4.5.3, CRESTA - Collaborative Research Into Exascale Systemware, Tools and Applications, 2013.

Model Checking with User-Definable Memory Consistency Models

Tatsuya Abe and Toshiyuki Maeda

RIKEN AICS, Kobe, Hyogo, Japan
{abet,tosh}@riken.jp

Abstract

From the viewpoint of performance and scalability, relaxed memory consistency models are common and essential for parallel/distributed programming languages in which multiple processes are able to share a single global address space, such as Partitioned Global Address Space languages. However, a problem with relaxed memory consistency models is that programming is difficult and error-prone because they allow non-intuitive behaviors that do not occur in the ordinary sequential memory consistency model.

To address the problem, this paper presents a model checking framework in which users are able to define their own memory consistency models, and check programs under the defined models. The key point of our model checking framework is that we define a base model that allows very relaxed behavior, and allow users to define their memory consistency models as constraints on the base model.

1 Introduction

One of the problems of Partitioned Global Address Space (PGAS) languages from the viewpoint of ease of writing programs is that they are based on relaxed memory consistency models [3], like multicore CPUs [12] and conventional distributed shared memory systems [14]. A memory consistency model is a formal model that defines the behavior of shared memory accessed simultaneously by multiple processes. In a relaxed memory consistency model, the shared memory behaves differently from that in a sequential one. More specifically, the behavior defined by a relaxed memory consistency model may not match any possible behavior of a sequential process that simulates the behavior of multiple processes by executing their instructions in an arbitrary interleaved way.

To prevent non-intuitive behaviors of relaxed memory models, programmers must insert explicit synchronization operations in their programs; however, this is difficult and error-prone. Conservatively inserting synchronization operations severely degrades the performance of the program. However, it is difficult to reduce the number of synchronization operations correctly because even a slight lack of synchronization can introduce unpredictable and/or non-reproducible bugs.

One possible approach to address the above mentioned problem of relaxed memory consistency models is to apply *model checking* to verify programs. Model checking is a formal program verification approach that explores all the possible program states that can be reached during program execution. Software model checkers are available for parallel/distributed programs including for PGAS languages [10, 17, 20, 7, 9, 1, 2].

However, most of the existing studies of model checking parallel/distributed programs consider the sequential memory consistency model only. Therefore, they cannot be used to verify programs executing on shared memory with relaxed memory consistency models. Although several studies of program verification considering relaxed memory consistency models have been carried out (e.g., [11, 5, 4, 6]), most of these support only one or a small number of fixed

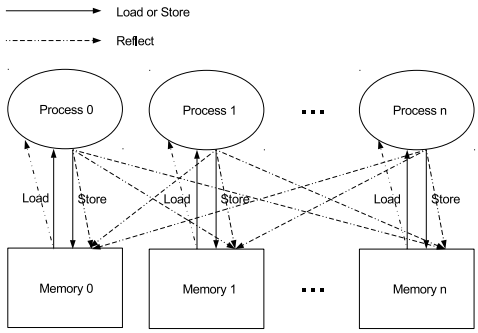


Figure 1: Overview of our abstract machine

theoretical models, and are thus not suitable for handling practical relaxed memory consistency models that may vary from language to language. Some of the works try to handle multiple relaxed memory consistency models uniformly (e.g., [18, 22, 21, 16, 8]), but these are still limited to a few existing relaxed memory consistency models and it is not apparent how to adapt and support the other models not covered by these.

To address the problem, this paper presents a model checking framework in which users are able to define their own memory consistency models, and verify programs under the defined models. The key point of our approach is that we define and provide a base model that allows very relaxed behaviors, and allows users to define their memory consistency models as constraints on the base model.

The rest of this paper is organized as follows. Section 2 describes our base model on which memory consistency models can be defined. Section 3 explains how to define memory consistency models using example constraint rules representing rules of memory consistency models for Coarray Fortran. Section 4 briefly introduces the implementation of our model checker. Finally, Section 5 concludes the paper and discusses future work.

2 Base Model

In this section, we introduce our base model on which various memory consistency models can be defined. More specifically, in Section 2.1 we introduce an abstract machine for the model. Then, we describe the execution traces of the abstract machine in Section 2.2.

2.1 Abstract Machine

An overview of our abstract machine is shown in Fig. 1. The abstract machine consists of a fixed number of processes. Each process has its own state (that is, program instructions, local variables, and memory). Basically, execution of the instructions is closed within each process, and communication between processes occurs only when a process performs a store operation on its memory, and the update is broadcast to the other processes.

A more formal definition of the abstract machine is given in Figs. 2 and 3. Owing to space limitations, the semantics of the abstract machine is not given in this paper. The semantics is standard and straightforward, except that the instructions can be reordered and memory store instructions send requests for memory updates to other processes. The next section describes

$$
\begin{array}{lll}
\text{(State)} & S & ::= (P_1,\ \ldots,\ P_n) \\
\text{(Process)} & P & ::= (I, V, M) \\
\text{(Store)} & V & ::= \{x_1 \mapsto v_1,\ \ldots,\ x_i \mapsto v_i\} \\
\text{(Memory)} & M & ::= \{\ell_1 \mapsto v_1,\ \ldots,\ \ell_j \mapsto v_j\} \\
\multicolumn{3}{l}{\quad \text{where}} \\
\text{(Variable)} & x & \quad \text{(local variables)} \\
\text{(Location)} & \ell & \quad \text{(addresses in memory)} \\
\text{(Label)} & L & \quad \text{(labels in instructions)} \\
\text{(Value)} & v & ::= n \mid \ell \mid L
\end{array}
$$

Figure 2: Definition of our abstract machine

$$
\begin{array}{lll}
\text{(Insts.)} & I & ::= i_1;\ \ldots;\ i_n \\
\text{(Inst.)} & i & ::= (L, A, \iota) \\
\text{(Raw Inst.)} & \iota & ::= \texttt{Move}\ x \leftarrow t \ \mid\ \texttt{Load}\ x \leftarrow [x] \\
 & & \quad \mid\ \texttt{Store}\ [x] \leftarrow t \ \mid\ \texttt{Jump}\ t_t\ \texttt{if}\ t_c \\
 & & \quad \mid\ \texttt{Atomic}\ (\iota_1;\ \ldots;\ \iota_n) \ \mid\ \texttt{Nop} \\
\text{(Term)} & t & ::= x \mid v \\
\text{(Attrs.)} & A & ::= \{a_1,\ \ldots,\ a_n\} \\
\text{(Attribute)} & a & \quad \text{(user-defined attributes)}
\end{array}
$$

Figure 3: Definition of the instructions

how to handle instruction reordering in our base model.

2.2 Execution Traces

This section explains how to handle relaxed memory consistency models in our base model; that is, how to simulate the effect that memory accesses performed by one process can be observed in a different order by other processes. The key ideas are threefold. First, our model decomposes an instruction into the fetch and issue of the instruction. Additionally, the issue of a memory instruction is further decomposed into the issue itself and its corresponding memory operation. Specifically, the issue of a memory load instruction is decomposed into the issue itself and the memory access, which stores the obtained value in a variable local to its own process. On the other hand, the issue of a memory store instruction is decomposed into the issue itself and memory updates on each process. Second, each process in the abstract machine executes not only its own instructions, but also the other processes instructions and memory operations. At first sight, this may appear redundant, but it is necessary to handle some relaxed memory consistency models that allow processes to observe inconsistent shared memory images. Third, our model considers all the possible permutations of instruction issues and memory operations performed by all the processes. In this paper, we call the permutations *execution traces* (or simply *traces*).

$$
\begin{array}{lll}
\text{(All Traces)} & T_S & ::= \{\tau \mid S \vDash \tau\} \\
\text{(Trace)} & \tau & ::= o_1;\ \ldots;\ o_n;\ \ldots \\
\text{(Operation)} & o & ::= \texttt{Fetch}_{p'}\ p\ i \\
 & & \quad \mid\ \texttt{Issue}^{m}_{p'}\ p\ i \\
 & & \quad \mid\ \texttt{Rflct}^{m}_{p'}[\Rightarrow p]\ i\ \ell\ v
\end{array}
$$

Figure 4: Definition of execution traces

A more formal definition of execution traces is given in Fig. 4. Trace τ is defined as an ordered (finite or infinite) sequence of operations o, where o is either an instruction fetch (denoted by the tag `Fetch`), an instruction issue (denoted by the tag `Issue`), or a memory operation (denoted by the tag `Rflct`). Please note that the subscript p' of the operation tags indicates that the operations are included in the execution of the abstract machine by process p'. (Recall that as mentioned above, each process executes not only its own instructions but also the instructions and memory operations of the other processes.) A more detailed explanation is omitted owing to space limitations.

3 Defining Consistency Models

In this section, we explain how to define memory consistency models on the base model described in Section 2, by showing an example rule that represents a barrier instruction of Coarray Fortran [15]. Informally speaking, memory consistency models are defined as constraints on execution traces, and the traces satisfying the constraints are considered to be valid under the memory consistency models.

Coarray Fortran has a barrier instruction called *sync all*, and the memory consistency model of Coarray Fortran [15] ensures that instructions that are fetched before (or after) the barrier instruction have to be completed before (or after) the barrier instruction completes. For example, the following rule preserves the order between the barrier instruction (sync all) and a store instruction:

$$
\begin{aligned}
&\forall p, p', i, i'. \\
&\qquad \texttt{Fetch}_{p'}\ p\ i \downarrow \texttt{Fetch}_{p'}\ p\ i' \supset \\
&\qquad\quad \forall p''.\ \texttt{Issue}^{m}_{p'}\ p\ i \downarrow \texttt{Rflct}^{m'}_{p'}\left[\Rightarrow p''\right]\ i'\ \ell\ v \\
&\quad \text{and } \texttt{Fetch}_{p'}\ p\ i' \downarrow \texttt{Fetch}_{p'}\ p\ i \supset \\
&\qquad\quad \forall p''.\ \texttt{Rflct}^{m'}_{p'}\left[\Rightarrow p''\right]\ i'\ \ell\ v \downarrow \texttt{Issue}^{m}_{p'}\ p\ i \\
&\text{if } i \equiv (\_, A, \texttt{Nop})\,,\ i' \equiv (\_, \_, \texttt{Store}\ [x] \leftarrow t)\,, \\
&\quad \text{and } \texttt{sync-all} \in A
\end{aligned}
$$

In the above rule, $o_1 \downarrow o_2$ means that o_1 appears before o_2 in execution traces. In addition, m and m' represent the positions of i and i' in the trace, respectively, $\ell = V(x)$, and $v = V(t)$, where V is the state of the local variables of process P_p when instruction i' is issued. Note that the attribute of instruction i is used to indicate that i is a barrier operation.

4 Implementation

Based on the approach explained in Sections 2 and 3, we implemented a prototype model checker. Basically, the model checker takes a user program and a memory consistency model as inputs, and generates a model in PROMELA, which is the model description language of the SPIN model checker [10]. Then, the generated model is passed to the SPIN model checker to perform model checking. Note that model checking is conducted in such a way that each process defined in the input program explores the generated model independently (refer to Section 2.2).

In fact, using the proposed model checker, we successfully checked several small example programs taken from the specification documents of UPC [19] and Itanium [13]. In addition, the model checker was used to analyze differences between three relaxed memory consistency models: UPC [19], Coarray Fortran [15], and Itanium [13].

5 Conclusion and Future Work

In this paper we described an approach to model checking parallel/distributed programs with shared memory (e.g., PGAS programs) under relaxed memory consistency models. In our approach, users are able to define memory consistency models as constraint rules on our quite relaxed base model. Based on this approach, we implemented a prototype model checker with which we conducted several preliminary experiments.

Future work includes two directions. First, we intend on improving the algorithm for model checking by utilizing, for example, partial order reduction. Because the current implementation of our model checker is prone to the state explosion problem, it is necessary to further optimize the algorithm to enhance its practicality. Second, we aim to develop a formal system for our base model based on Kripke semantics to verify properties directly by theorem proving.

References

[1] T. Abe, T. Maeda, and M. Sato. Model checking with user-definable abstraction for partitioned global address space languages. In *Proceedings of the 6th Conference on Partitioned Global Address Space Programming Models*, 2012.

[2] T. Abe, T. Maeda, and M. Sato. Model checking stencil computations written in a partitioned global address space language. In *Proc. of the 18th International Workshop on High-level Parallel Programming Models and Supportive Environments*, pages 365–374, Cambridge, May 2013.

[3] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.

[4] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. *SIGPLAN Not.*, 45(1):7–18, Jan. 2010.

[5] G. Boudol and G. Petri. Relaxed memory models: an operational approach. *SIGPLAN Not.*, 44(1):392–403, Jan. 2009.

[6] A. Dan, Y. Meshman, M. Vechev, and E. Yahav. Predicate Abstraction for Relaxed Memory Models. In *Proc. of Static Analysis Symposium*, 2013.

[7] A. Ebnenasir. UPC-SPIN: A framework for the model checking of UPC programs. In *Proceedings of the 5th Partitioned Global Address Space Conference*. ACM, 2011.

[8] R. Ferreira, X. Feng, and Z. Shao. Parameterized Memory Models and Concurrent Separation Logic. *European Symposium on Programming*, 6012:267–286, 2010.

[9] M. Gligoric, P. C. Mehlitz, and D. Marinov. X10X: Model checking a new programming language with an "old" model checker. In *Proc. of International Conference on Software Testing, Verification, and Validation*, pages 11–20, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[10] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[11] T. Huynh and A. Roychoudhury. A memory model sensitive checker for c#. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 476–491. Springer Berlin Heidelberg, 2006.

[12] Intel Corporation. *Intel Itanium Architecture Software developer's manual*.

[13] Intel Corporation. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*, 2010.

[14] D. Mosberger. Memory consistency models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, 1993.

[15] J. Reid and R. W. Numrich. Co-arrays in the next Fortran standard. *Scientific Programming*, 15(1):9–26, 2007.

[16] V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A Theory of Memory Models. In *Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 161–172, New York, New York, USA, Mar. 2007. ACM Press.

[17] S. F. Siegel. Model checking nonblocking MPI programs. In *Proc. of International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 44–58, Jan. 2007.

[18] R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5):800–849, Sept. 2004.

[19] The UPC Consortium. *UPC Language Specifications V1.2*, 2005.

[20] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP: a tool for model checking MPI programs. In *Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 285–286, New York, New York, USA, Feb. 2008. ACM Press.

[21] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Memory-model-sensitive data race analysis. In *Proc. of International Conference on Formal Methods and Software*, pages 30–45, 2004.

[22] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos : A Framework for Axiomatic and Executable Specifications of Memory Consistency Models. In *Proc. of International Parallel and Distributed Processing Symposium*, 2004.

Static Analysis for Unaligned Collective Synchronization Matching for OpenSHMEM

Swaroop Pophale[1], Oscar Hernandez[2], Stephen Poole[2] and Barbara Chapman[1]

[1] University of Houston Houston, Texas, USA
spophale,chapman@cs.uh.edu
[2] Oak Ridge National Lab Oak Ridge, Tennessee, USA
oscar, spoole@ornl.gov

Abstract

The most critical step for static concurrency analysis of OpenSHMEM is to detect matching unaligned collective synchronization calls within an OpenSHMEM program. Concurrency analysis will be able to detect the regions in the code where two or more OpenSHMEM calls run concurrently and help identify parallel programming errors due to incorrect usage of OpenSHMEM library calls. For the concurrency analysis to be accurate the collective synchronization matching process must be accurate. This task is particularly challenging for OpenSHMEM programs since the OpenSHMEM library provides textually unaligned *barriers* over an ***active set*** in addition to the traditional *barrier_all* statement. An active set is essentially a logical grouping of processing elements. In this paper we discuss our effort towards discovering and matching *barrier* calls in OpenSHMEM. We extend the OpenSHMEM Analyzer (OSA) to discover potential synchronization errors due to unaligned barriers in OpenSHMEM programs.

1 Introduction

Parallel programming libraries are critical for High Performance computing applications. OpenSHMEM is a PGAS library that can be employed on shared as well as distributed systems to aid SPMD programs to achieve potentially low-latency communication via its *onesided* data transfer calls. Other than data transfer an OpenSHMEM library provides library calls for collective operations (broadcast, reductions, collections and synchronization), atomic memory operations, distributed locks and data transfer ordering primitives (fence and quiet). As all libraries go, OpenSHMEM is limited by the lack of compiler support to ensure the correct use of the library in a parallel context. A part of the burden is alleviated by the syntactic and basic semantic checks already present in the OpenSHMEM Analyzer (OSA) [10]. OSA extends the existing compiler technology to report errors accurately in context of C and OpenSHMEM. In this paper we extend the OSA to perform barrier matching analysis and make the control flow graph (CFG) aware of OpenSHMEM calls and its the SPMD semantics [11]. Studies have determined that *multi-valued expressions* that affect control flow statements the cause of concurrency [8,9,13] and are needed for barrier matching analysis. Multi-valued expressions are those that evaluate to different values on different processes [3]. A *multi-valued expression* depends on a multi-valued seed/variable. A generic example of multivalued seed is the process id (which we know is unique for different processes). Another level of complexity exists for OpenSHMEM where the the collective synchronization statements are textually unaligned and may be applicable only over a portion of the total processing elements (PEs) executing the same code. Barrier matchng [15] across the different concurrent regions within an OpenSHMEM program is needed to detect possible synchronization errors. We present a framework[1] for modifying

[1]This work is supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center located at the Oak Ridge National Laboratory.

and analyzing the CFG to build a program dependence graph and perform unaligned barrier matching in presence of different collective synchronization calls, namely, *shmem_barrier* (which is over an *active set*) and *shmem_barrier_all*.

2 Related Work

Collective synchronization matching has a myriad of applications. It is used as a stepping stone to facilitate complex analysis for process-level parallelism analysis of a program in [6] . Their analysis avoids the problem of having to identify textually unaligned barriers by assuming that barriers are identified via unique barrier variables. One of the first works to verify program synchronization patterns and the rules that govern the synchronization sequences was done in [2] for Split-C. They analyze the effects of *single valued expression* on the control flow and concurrency characteristics of the program. They simplify the identification of unaligned barriers and single valued variables by using the *single* keywords for annotating the named barriers. Barrier matching for MPI [15] evaluates the different concurrent paths the processes may take (using multi-value conditional and barrier expression analysis) and check that each processes encounters an equal number of barriers. Our approach to barrier matching is similar but we distinguish our approach by avoiding the use of barrier subtrees to match barriers. This makes our analysis more resilient to unstructured code.

3 Motivation

OpenSHMEM is a PGAS library that provides routines for programmers using the SPMD programming paradigm. The OpenSHMEM Specification [1] provides the definition and functionality of these concise and powerful library calls for communicating and processing data. We first describe the collective calls and then discuss their implication and potential error scenarios they could lead to where programs may be syntactically correct but are either semantically wrong or may result in parallel behavior unintended by the programmer. Collective synchronization in OpenSHMEM is provided by *shmem_barrier* and *shmem_barrier_all* (over a subset of PEs and all PEs respectively) in OpenSHMEM.

3.0.1 shmem_barrier_all

A *shmem_barrier_all* (referred to as *barrier_all*) is defined over all PEs. OpenSHMEM requires all PEs to call *shmem_barrier_all* at the same point in the execution path. It provides global synchronization and its semantics guarantee completion of all local and remote memory updates once a PE returns from the call.

3.0.2 shmem_barrier

A *shmem barrier* (referred to as *barrier*) is defined over an ***active set***. An *active set* is a logical grouping of PEs based on three parameters (passed as arguments), namely,***PE_start***, ***logPE_pe*** and ***PE_size*** triplet [1]. OpenSHMEM requires all PEs within the *active set* to call *shmem_barrier* at the same point in the execution path. When barriers are not matched it is a obvious dead-lock situation, but even in cases where all barrier (barrier all and barrier) statements are well matched the compiler needs to differentiate between the two to make sure that the semantics of the OpenSHMEM library are not violated. In the code below we observe one such situation.

```
1  start_pes(0);
2  int me = _my_pe();
3  ...
4  if(me%2 == 0){
5    source = 0.23 * me;
6    shmem_int_put(&target, &source, 1,(me+1)% npes);
7    shmem_barrier(0,1,4,pSync); //b1
8    x = target;
9    shmem_barrier_all(); //b2
10 }
11 else{
12   x = target ;
13   shmem_barrier_all(); //b3
14 }
15 return 0;
```

OpenSHMEM_example.c

In this example all barrier statements are matched. We see that on line 5 all even numbered PEs update their value of the variable *source* and immediately after, update the value of the *symmetric* variable *target* on the next PE in a circular fashion. Simultaneously all odd numbered PEs are updating a local variable x with the value in their *symmetric* variable *target* (line 12). Since the semantics of OpenSHMEM only guarantee the completion of *puts* after synchronization, on line 12, the odd numbered PEs may or may not have the updated value of *target*. The programmer may easily miss such errors leading to an application with inconsistent results. For this type of analysis barrier matching is essential to detect concurrent execution phases for reads and writes to the same variables along with data flow analysis.

4 Unaligned Collective Synchronization Matching Framework

To be able to discover matching barriers in OpenSHMEM program we need to first locate and mark the **multi-valued expressions** which cause different processes to follow different execution paths. **Multi-valued** expressions evaluate to different results on different PEs. The outcome of a multi-valued expression depends on a **multi-valued** seed. Extending the rules stated in [3] we can make certain assumptions about the expressions that generate from a known single-valued or multi-valued seed. Depending on the semantics of OpenSHMEM and its treatment of different program variables, the rules for determining multi-valued seed have to be modified. Different library calls cause changes in the values that may make them *single* or *multi-valued*. This information is required for a finer analysis and to avoid overestimation of concurrency. Table1 lists the OpenSHMEM library operation categories that cause a change in the value of a program variable (this includes all variables used in the program irrespective of their class) and the effect they have on the variable. Explanation of the API and the nomenclature is beyond the scope of the paper (refer to OpenSHMEM Specification 1.0 [1]).

4.1 Identifying Multi-value Conditionals

To identify multi-value conditionals we need to essentially follow the data flow propagation of the multi-value seeds through the program's CFG and mark the conditionals that are affected directly or indirectly by them [15]. A *system dependence graph* [5] needs to be built. We look at program slicing as a way to identify the reach of these multi-valued seeds. Programming slicing is defined as, "A decomposition based on data flow and control flow analysis" [14]. To

OpenSHMEM Library Operations	Variable Affected	Effect
_num_pes	*npes*	Single-valued
_my_pe	*me*	Multi-valued
PUT (elemental, block, strided)	*target*	Multi-valued
GET (elemental, block, strided)	*target*	Multi-valued
ATOMICS (fetch and operate)	*target*	Multi-valued
BROADCAST	*target*	Multi-valued
COLLECTS (fixed and variable length)	*target* array	Single-valued if *active set* = npes else Multi-valued

Table 1: **Effect of OpenSHMEM library calls on program variables**

do the program slicing, we look at the data flow and control flow information generated by the compiler and build a *system dependence graph* [5] as shown in Figure 1. If we were to take a forward slice of the sample program based on the *multi-valued* PE number *me* at A2, then we get either A3-B1-B2-B3-B4-B5-D or A3-C1-C2-D depending on the value of *me*. These slices help us identify the *multi-valued* conditionals in the program by finding the points at with the slices diverge.

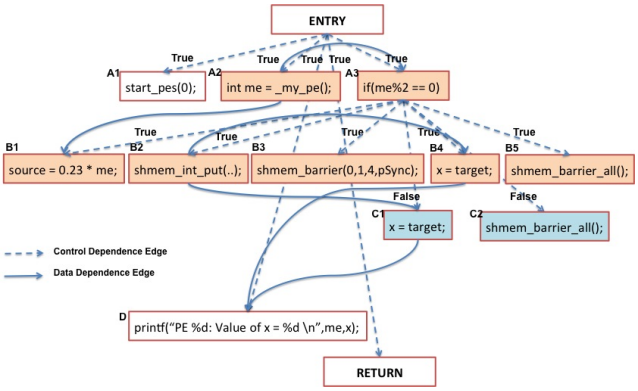


Figure 1: System Dependence Graph

4.2 Synchronization Matching

To facilitate analysis between and across barrier regions we need to identify *code phases* (a valid synchronization free path enclosed within barriers) that lie between matching barriers. We use the approach in [15] for developing barrier expressions and barrier trees as the first step towards barrier matching. As defined in [15] *barrier expressions* are very similar to path expressions and can be generated from them by replacing the node labels by barrier labels. Like regular expressions these expressions use three types of operators: concatenation ($\cdot$), alternation ($|$), and quantification ($*$) [7]. Additionally we borrow the operator $|^c$ from [15] to indicate the operator *concurrent alternation*, which essentially indicates that the different execution paths diverge

Placement of barriers	Operator used	Result
b1 followed by b2	$\cdot$	b1 $\cdot$ b2
if((*single-valued*) *conditional*) b1 else b2;	|	b1 | b2
if((*multivalued-valued*) *conditional*) b1 else b2;	$\|^c$	b1 $\|^c$ b2
for(n times) b;	$\cdot$	b1$\cdot$ b2 $\cdot$ ... bn

Table 2: **Rules for building a barrier expression**

from a *multi-valued* conditional and that different PEs may take different paths from this point on. Table 2 gives the rules that govern the barrier expression generation. It is important to note that if the result of a quantification operation is statically non-deterministic (like a barrier enclosed within a while loop based on a value available only at run-time) we cannot determine with confidence any concurrency relationship for such programs. Using the operators mentioned above we can derive the *barrier expression* and the *barrier tree*. The barrier expressions can be generated by first generating the path expressions using methods in [4] or [12]. The barrier expression for the entire program is generated by first evaluating the barrier expressions of the individual procedures and then connecting them together using the inter-procedural program dependence graph. Barrier matching is done by first generating the legal sequences of barriers by a constraint driven depth-first-search of the barrier tree and validating barrier sequences that have at least one other barrier sequence of the same length.

5 Evaluation

As of this writing we are testing our methodology using the OpenSHMEM versions of NAS Parallel benchmarks (in C). We have been successful in detecting matching barriers in all the smaller test codes provided with the OpenSHMEM Validation and Verification Suit. For some codes we intentionally added unmatched barriers and our algorithm was able to detect it. In severely unstructured codes or with conditional values which are runtime dependent (which are defaulted to multivalued) our algorithm generates a warning even if barriers are matched.

References

[1] Openshmem specification.

[2] Alexander Aiken and David Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 342–354, New York, NY, USA, 1998. ACM.

[3] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pages 149–159, New York, NY, USA, 1996. ACM.

[4] Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *J. ACM*, 23(1):172–202, January 1976.

[5] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[6] Tor E. Jeremiassen and Susan J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *Proceedings of the IFIP WG10.3*, PACT '94, pages 171–180, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[7] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, 1956.

[8] Yuan Lin. Static nonconcurrency analysis of openmp programs. In *Proceedings of the 2005 and 2006 IWOMP*, pages 36–50, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 129–138, New York, NY, USA, 1993. ACM.

[10] Hernandez Oscar, Jana Siddhartha, Swaroop Pophale, Poole Stephen, Jeffery Kuehn, and Chapman Barbara. The openshmem analyzer. In *Proceedings of the Sixth Conference on PGAS Programming Model*, New York, NY, USA, 2012. ACM.

[11] Michelle Mills Strout, Barbara Kreaseck, and Paul D. Hovland. Data-flow analysis for mpi programs. *2012 41st ICPP*, pages 175–184, 2006.

[12] Robert Endre Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, July 1981.

[13] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Commun. ACM*, 26(5):361–376, May 1983.

[14] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[15] Yuan Zhang and Evelyn Duesterwald. Barrier matching for programs with textually unaligned barriers. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 194–204, New York, NY, USA, 2007. ACM.

Performance Evaluation of Unified Parallel C for Molecular Dynamics

Kamran Idrees[1], Christoph Niethammer[1], Aniello Esposito[2] and Colin W. Glass[1]

[1] High Performance Computing Center Stuttgart (HLRS), Stuttgart, Germany
idrees@hlrs.de, niethammer@hlrs.de, glass@hlrs.de
[2] Cray Inc., Seattle, WA, U.S.A.
esposito@cray.com

Abstract

Partitioned Global Address Space (PGAS) integrates the concepts of shared memory programming and the control of data distribution and locality provided by message passing into a single parallel programming model. The purpose of allying distributed data with shared memory is to cultivate a locality-aware shared memory paradigm. PGAS is comprised of a single shared address space, which is partitioned among threads. Each thread has a portion of the shared address space in local memory and therefore it can exploit data locality by mainly doing computation on local data.
Unified Parallel C (UPC) is a parallel extension of ISO C and an implementation of the PGAS model. In this paper, we evaluate the performance of UPC based on a real-world scenario from Molecular Dynamics.

1 Introduction

Partitioned Global Address Space (PGAS) is a locality-aware distributed shared memory model for Single Program Multiple Data (SPMD) streams. PGAS unites the concept of shared memory programming and distributed data. It provides an abstraction of Global Shared Address Space, where each thread can access any memory location using a shared memory paradigm. The Global Shared Address Space is formed by integrating the portions of the memories on different nodes and the low level communication involved for accessing remote data is hidden from the user. Unified Parallel C (UPC) is an implementation of the PGAS model. The low-level communication in UPC is implemented using light-weight Global-Address Space Networking (GASNet). UPC benefits from the brisk one-sided communication provided by GASNet and thus has a performance advantage over message passing [5].

Molecular Dynamics simulates the interactions between molecules [1]. After the system is initialized, the forces acting on all molecules in the system are calculated. Newton's equations of motion are integrated to advance the positions and velocities of the molecules. The simulation is advanced until the computation of the time evolution of the system is completed for a specified length of time.

In this paper, we evaluate the intra- and inter-node performance of UPC based on a real-world application from Molecular Dynamics, compare it intra-node with OpenMP and show the necessity for manual optimizations by the programmer in order to achieve good performance.

2 Unified Parallel C

Unified Parallel C (UPC) is a parallel extension of ISO C. It is a distributed shared memory programming model that runs in a SPMD fashion, where all threads execute the main program or function. Using UPC constructs, each thread can follow a distinct execution path to work on

different data. UPC threads run independently of each other, the only implied synchronization is at the beginning and at the end of the main function [4]. It is the responsibility of the programmer to introduce necessary synchronization when shared data is accessed by more than one thread. Apart from a global shared address space, UPC also provides private address space for each thread. The private address space is only accessible by the thread whichs owns it. This allows a programmer to intelligently allocate the data in private and shared address spaces. Data which remains local to a thread should be allocated on the private address space. Whereas data which needs to be accessed by multiple UPC threads, should be allocated on the portion of the shared address space of the thread doing most computation on it [4].

UPC accommodates several constructs which allow to allocate data and the thread with affinity to it on the same physical node. UPC also provides constructs to check the locality of data. The programmer needs to identify data as local in order to access it with a local pointer.

UPC utilizes a source to source compiler. The source to source compiler translates UPC code to ANSI C code (with additional code for communication to access remote memory, which is hidden from the user) and links to the UPC run-time system. The UPC run-time system can examine the shared data accesses and perform communication optimizations [5] [4].

3 Molecular Dynamics Code

We ported our in-house Molecular Dynamics code CMD, developed for basic research into high performance computing. CMD features multiple MD data structures, algorithms and parallelization strategies and thus allows for quantitative comparisons between them. Two widely used data structures are implemented - with corresponding algorithms - for the computation of interactions between molecules in the system, “BasicN2” and “MoleculeBlocks”. The MoleculeBlocks code has been ported to UPC.

MoleculeBlocks implements a linked cell approach, where the domain is spatially decomposed into cells (of the size cut-off radius) and then the molecules are distributed among these cells. In this algorithm, the distances between the molecules are computed only intra-cell and for neighboring cells. Furthermore, Newton’s 3rd law of motion is used to reduce the compute effort by half. Figure 1 shows an example of the MoleculeBlocks algorithm for a 2D domain space. When the interaction between a pair of molecules is computed, the resulting force is written to both molecules. Thus, the centered cell (dark gray), as shown in figure 1, modifies the forces of its own molecules and molecules of its right and lower neighbor cells (gray). Although the use of Newtons 3rd law lessens the computational effort, it raises the requirements regarding synchronization in order to avoid race conditions.

4 Porting Molecular Dynamics Code to UPC

For MoleculeBlocks, the system of molecules (called Phasespace) is spatially decomposed into cells where each cell contains a number of molecules (as shown in figure 2). The cells are then distributed among the UPC threads in a spatially coherent manner (as opposed to the default round-robin fashion) to reduce the communication overhead between the UPC threads. The CMD simulation code is comprised of two parts, (i) phasespace initialization & grid generator and (ii) main simulation loop.

4.1 Phasespace Initialization & Grid Generator

Phasespace initialization involves allocation of the memory dynamically on the global shared space for molecules and cells. A routine for performing the transformation (or mapping) of a spatially coherent cell indices (i,j,k) to consecutive integers (cell IDs) is introduced in the

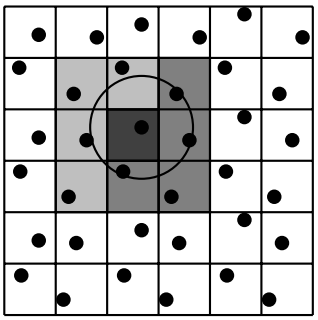


Figure 1: Calculation of interaction between molecules using the `MoleculeBlocks` algorithm.

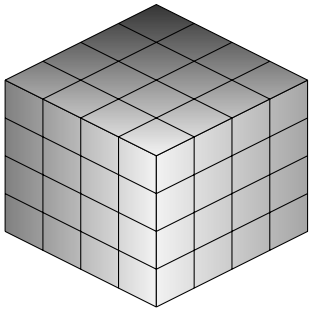


Figure 2: Domain is spatially decomposed into cells and distributed among threads in a spatially coherent manner.

code, which allows UPC shared array to distibute spatially coherent cells among UPC threads by blocking consecutive cell IDs. The Grid generation routine initializes the positions and velocities of the molecules and adds them to the phasespace.

4.2 Main Simulation Loop

All the routines inside the main simulation loop have been implemented in UPC as locality-aware algorithms. Each thread only calculates the different parameters for the cells which reside in its portion of shared space. Due to manual optimization, each thread accesses its cells using local pointer instead of pointer-to-shared.

The Lennard-Jones Force Calculation routine is the most compute intensive routine of the simulation. Here, each thread computes the inter-molecular interactions, for the cells residing in its portion of shared space and with the neighbor cells which may reside locally or remotely. The synchronization among threads is provided through a locking strategy. We have tested different locking granularities, which are "lock per molecule" and "lock per cell". Furthermore, we have also implemented pre-fetching and copy-at-once strategies to reduce the communication between UPC threads. This has a major effect when the UPC code is scaled beyond a single node. With pre-fetching and copy-at-once strategies, a UPC thread pre-fetches the values (of positions and forces) of the molecules of its neighbor cell if the neighbor cell is remote (i.e. does not reside in its local portion of the shared space) to its private space. The function of pre-fetching is implemented using the upc_memget routine. Once a UPC thread has pre-fetched the data, it computes all interactions between the molecules of the two cells and then copies all the calculated forces to the neighbor cell in one go using the upc_memput routine.

In order to calculate the global value of parameters (e.g potential energy), the coordination among threads is done using the reduction function upc_all_reduceT available in the collective library of UPC.

5 Hardware Platforms

The presented benchmarks have been produced on a Cray XE6 system and a Cray XC30 system. The code was built by means of the Berkeley UPC compiler (version 2.16.2) on both systems. The cray compiler had some performance issues which are under investigation. The nodes of the XE6 system feature two AMD Interlagos processors (AMD Opteron(TM) Processor 6272 clocked at 2.10GHz) with 16 integer cores each. Two integer cores share a floating

point unit. On the other hand, the compute nodes of the XC30 system contain two Intel Sandybridge processors (Intel(R) Xeon(R) CPU E5-2670 0 clocked at 2.60GHz) with 8 cores and 8 hyperthreads each. For the present benchmarks no hyperthreads have been used.

For a full documentation and technical specifications of the hardware platforms, the reader is referred to the online material[1].

6 Evaluation

The UPC implementation of CMD is evaluated for the cut-off radius of 3 with the following strategies (a lock always locks the entire cell).

1. *Lock per molecule (LPM)* - Acquire lock for each molecule-molecule interaction
2. *Lock per cell (LPC)* - Acquire lock for each cell-cell interaction
3. *Lock per cell plus prefetching (LPC+)* - Same as lock per cell but with pre-fetching and copy-at-once strategies

The cut-off radius determines the maximum distance for evaluating molecule-molecule interactions. Increasing the cut-off will result in more interactions per molecule and therefore more computational effort. The cell size is equal to the cut-off radius.

In the first UPC implementation of CMD, we did not perform the manual pointer optimizations. The shared address space was always accessed using pointer-to-shared irrespective of the fact whether the data accessed by a UPC thread is local or remote. The test cases of 500, 3,000 and 27,000 molecules without pointer optimizations executed 10 times slower than the current version which incorporates manual pointer optimizations.

The rest of the evaluation is based on the version with manual optimization, a cut-off radius of 3 and with 6,000,000 molecules. Thus, all scaling benchmarks shown here are based on strong scaling. The evaluation metrics are explained in the following subsection.

6.1 Evaluation Metrics

The UPC implementation of CMD is evaluated for all three strategies described above, on the basis of intra- and inter- node performance. For each case, we have taken the average execution time for five program runs.

Intra-Node Performance The intra-node performance compares the UPC and OpenMP implementations of CMD on a single node.

Inter-Node Performance The inter-node performance shows the scaling on multiple nodes. Under populated nodes are used for inter-node results when the number of UPC threads are less than the total available CPU count of 4 nodes. The threads are always equally distributed among the nodes.

6.2 Results

Here we show the execution time of CMD both intra- and inter-node, and compare the execution time with varying locking granularities.

Intra-Node Performance Figure 4 shows the intra-node benchmark for the execution time of UPC implementation of CMD. Clearly, the lock per cell strategy is superior to the lock per molecule. Pre-fetching and copy-at-once has no significant impact on intra-node performance. Figure 3 compares intra-node performance achieved with OpenMP and UPC. UPC performs similarly to OpenMP on a single node. This is a satisfactory result, as the aim is not to provide a better shared memory parallelization, but to use a shared memory paradigm for distributed memory. Having a comparable performance as OpenMP is a good basis.

[1] www.cray.com/Products/Computing/XE/Resources.aspx, www.cray.com/Products/Computing/XC/Resources.aspx

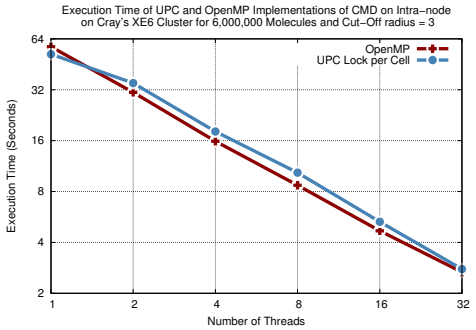


Figure 3: Comparison of intra-node execution time of UPC and OpenMP implementations of CMD on a Cray XE6.

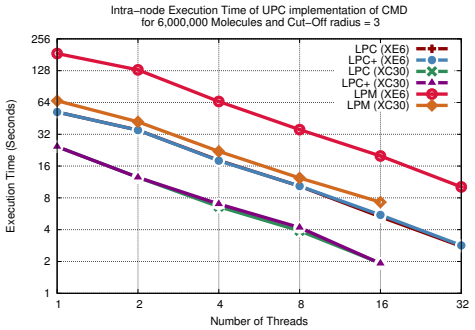


Figure 4: Intra-node execution time of UPC implementation of CMD with different synchronization strategies.

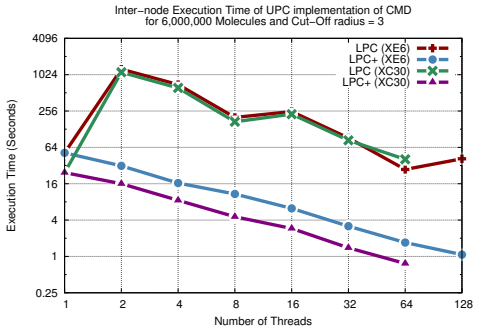


Figure 5: Inter-Node Execution Time of the UPC implementation with different locking strategies and cut-off radius of 3.

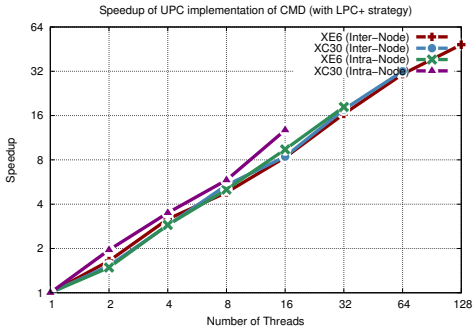


Figure 6: Speedup of UPC implementation of CMD with Lock per cell plus prefetching strategy on intra- and inter-node.

Inter-Node Performance Figure 5 shows the inter-node performance achieved with UPC, using LPC and LPC+. The LPM strategy is disregarded due to its inferior intra-node performance. As can be seen, the LPC strategy shows very poor inter-node performance (2 or more threads). The execution time jumps by a factor of 20+, as soon as inter-node communication comes into play. However, the LPC+ strategy shows a solid scaling behaviour, making this implementation competitive even for production runs. Figure 6 shows the speedup of CMD with LPC+ strategy on intra- and inter-node, for both XE6 and XC30 clusters.

7 Conclusion

As we have shown in this paper, it is possible to implement a competitive distributed memory parallelization using UPC. However, the implementation is far from trivial. There are many pitfalls, leading to significant performance degradations and they are not always obvious.

The first and most elusive problem is the use of pointer-to-shared for local data accesses. As a programmer, one would expect UPC compilers or the run-time to automatically detect local

data accesses and perform the necessary optimization. However, this is not the case and the performance degradation for our use case was a staggering factor 10. This suggests that with the currently available compilers, manual pointer optimization (using local C pointers when a thread has affinity to the data) is mandatory.

The second issue is not discussed in detail here. In a nutshell: the default round robin distribution of shared array elements leads to significant communication traffic in this scenario. Manual optimization was necessary, essentially replacing round robin with a spatially coherent distribution. Thus, the programmer needs to keep the underlying distributed memory architecture in mind.

The third and maybe most disturbing problem is related to communication granularity. The LPM and LPC strategies represent the way one traditionally would approach a shared memory parallelization. As the data is available in shared memory, there is no need to pre-fetch it or to package communication. However, these approaches fail completely when utilized for distributed memory, as can be seen in figure 5.

The good news is, all the above problems can be solved, the bad news is: it requires the programmer to think in terms of distributed memory parallelization. However, this is not the driving idea behind PGAS.

In our view, in order for PGAS approaches to prosper in the future, these issues have to be addressed. Only if better data locality, data distribution and communication pooling is provided automatically by the compiler or run-time will programmers start seeing true benefit. The required information for such improved automatic behaviour is available to the compiler, as the parallelization is achieved purely through UPC data structures and routines.

Acknowledgments This work was supported by the project *HA* which is funded by the German Research Foundation (DFG) under the priority programme "Software for Exascale Computing - SPPEXA" (2013-2015) and the EU project *APOS* which was funded as part of the European Commissions Framework 7.

References

[1] Michael P Allen. Introduction to molecular dynamics simulation. *Computational Soft Matter: From Synthetic Polymers to Proteins*, 23:1–28, 2004.

[2] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47. ACM, 2005.

[3] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[4] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: distributed shared memory programming*, volume 40. Wiley-Interscience, 2005.

[5] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, PASCO '07, pages 24–32, New York, NY, USA, 2007. ACM

The GASPI API specification and its implementation GPI 2.0

Daniel Grünewald[1] and Christian Simmendinger[2]

[1] Fraunhofer ITWM
Fraunhofer Platz 1, 67663 Kaiserslautern, Germany
daniel.gruenewald@itwm.fraunhofer.de
[2] T-Systems Solutions for Research
Pfaffenwaldring 38-40, 70569 Stuttgart, Germany
christian.simmendinger@t-systems.com

Abstract

GASPI (Global Address Space Programming Interface) is an API specification for Partitioned Global Address Spaces. The GASPI API is focused on three key objectives: scalability, flexibility and failure tolerance. GASPI uses one-sided RDMA driven communication in combination with remote completion in a PGAS environment. As such, GASPI aims to initiate a paradigm shift from bulk-synchronous two-sided communication patterns towards an asynchronous communication and execution model. GASPI follows a single program multiple data (SPMD) approach and offers a small, yet powerful API composed of synchronization primitives, fine-grained control over one-sided read and write communication primitives, synchronous and asynchronous collectives, global atomics, passive receives, communication groups and communication queues. With GPI 2.0, the GASPI standard has been implemented by Fraunhofer ITWM as a highly efficient open source implementation under GPL v3.

1 Introduction

As the supercomputing community prepares for the era of exascale computing, there is a great deal of uncertainty about viable programming models for this new era. HPC programmers will have to write application codes for systems which are hundreds of times larger than the top supercomputers of today. It is unclear whether the two-decades-old MPI programming model [5], by itself, will make that transition gracefully. Despite recent efforts to support true asynchronous communication, the message passing standard of MPI still focuses on bulk-synchronous communication and two-sided semantics to a large extent. Elapsed time for collective bulk-synchronous communication potentially scales with the logarithm of the number of processes, whereas the work assigned to a single process potentially scales with a factor of 1/(number of processes). Hence, the scalability of bulk-synchronous communication patterns appears to be limited at best. With todays ever increasing number of processes, a paradigm shift from bulk-synchronous communication towards an asynchronous programming model seems to be inevitable. While recent efforts in MPI 3.0 have improved support for one-sided communication and an asynchronous execution model, MPI still requires explicit exposure and access epochs with a corresponding entry and exit, i.e. a so-called PSCW cycle (Post,Start,Complete,Wait). We here present an alternative to the programming model of MPI. GASPI is a Partitioned Global Address Space (PGAS) API. In contrast to MPI, GASPI leverages one-sided RDMA driven communication with remote completion in a Partitioned Global Address Space. In GASPI data may be written asynchronously and one-sided, whenever it is produced, along with a corresponding notification. On the receiving side GASPI guarantees

that data is locally available whenever this notification becomes locally visible. The notification mechanism in GASPI enables its users to reformulate existing bulk-synchronous MPI applications towards an asynchronous dataflow model with a very fine-grained overlap of communication and computation. GASPI leverages zero-copy asynchronous writes and reads with RDMA queues and aims at a minimum of communication overhead. GASPI allows for a lot of flexibility in using its partitioned global address spaces (which in GASPI are called segments), but does not enforce a specific memory model. Implementing e.g the equivalent of shmem malloc/put/get functionality on top of the GASPI segments is rather straightforward. We have however opted against including this functionality in the core API since an enforcement of a symmetric global memory model does not fit irregular problems or heterogeneous hardware. GASPI allows its users to span multiple segments with configurable sizes and configurable participating ranks. GASPI also supports a variety of devices for its segments, like e.g GPGPU memory, main memory of Xeon Phi cards, main memory of host nodes or non-volatile memory. All these segments can directly read/write from/to each other within the node and across all nodes. With a growing number of nodes, failure tolerance becomes a major issue as machines expand in size. On systems with large numbers of processes, all non-local communication should be prepared for a potential failure of one of the communication partners. GASPI features timeouts for non-local functions, allows for shrinking or growing node sets and enables applications to recover from node-failures. In contrast to other efforts in the PGAS community, GASPI is neither a new language (like e. g. Chapel from Cray [1], UPC [2] or Titanium [6]), nor an extension to a language (like e. g. Co-Array Fortran [8]). Instead - very much in the spirit of MPI - it complements existing languages like C/C++ or Fortran with a PGAS API which enables the application to leverage the concept of the Partitioned Global Address Space. While other approaches to a PGAS API exist (e.g. OpenSHMEM [9] or Global Arrays), we believe that GASPI has a rather unique feature set like the support for multiple (and freely configurable) segments, failure tolerance or the concept of remote completion. A concept similar to remote completion exists in the ParalleX project. In principle, one may identify the triggering of a GASPI remote notification after delivering the corresponding data payload as the most simple of all possible ParalleX parcels.

2 GASPI overview

2.1 History

GASPI inherits much of its design from the Global address space Programming Interface (GPI [7, 3]), which has been developed in 2005 at the Competence Center for High Performance Computing (CC-HPC) at Fraunhofer ITWM. GPI is implemented as a low-latency communication library and is designed for scalable, real-time parallel applications running on cluster systems. It provides a PGAS API and includes communication primitives, environment run-time checks and synchronization primitives such as fast barriers or global atomic counters. GPI has been used to implement and optimize CC-HPC industry applications like the Generalized Radon Transform (GRT) method in seismic imaging or the seismic work flow and visualization suite PSPRO. Today, GPI is installed on Tier 0 supercomputer sites in Europe, including the HLRS in Stuttgart and the Juelich Supercomputing Centre. In 2010 the request for a standardization of the GPI interface emerged, which ultimately lead to the inception of the GASPI project in 2011. The work was funded by the German Ministry of Education and Science and included project partners Fraunhofer ITWM and SCAI, T-Systems SfR, TU Dresden, DLR, KIT, FZJ, DWD and Scapos. On June 14th 2013 the members of the GASPI project have released the

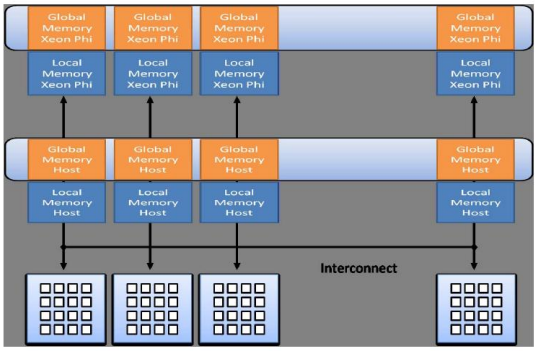


Figure 1: Two GASPI memory segments, one on Intel Xeon Phi, one on the x86 Host.

first specification for the new API [4]. At the same time, Fraunhofer ITWM has released GPI 2.0 [3], the first implementation of the GASPI specification.

3 Key concepts in GASPI

3.1 GASPI execution model

GASPI follows a SPMD (Single Program, Multiple Data) style in its approach to parallelism. Hence, a single program is started and initialized on all desired target computational units. GASPI adheres to the concept of ranks. Similarly to MPI each GASPI process receives a unique rank with which it can be identified during runtime. The GASPI API has been designed to coexist with MPI and hence in principle provides the possibility to complement MPI with a partitioned global address space. GASPI aims at providing interoperability with MPI in order to allow for incremental porting of existing applications. GASPI provides high flexibility in the configuration of the runtime parameters for the processes and allows for a shrinking or growing process set during runtime. In case of starting additional GASPI processes, the additional GASPI processes have to register with the existing GASPI processes.

3.2 GASPI groups

Groups are sub-sets of processes identified by a sub-set of the total set of ranks and closely related to the concept of a MPI communicator. Collective operations are restricted to the ranks forming the group. Each GASPI process can participate in more than one group. In case of failure, where one of the GASPI processes included within a given group fails, the group has to be reestablished.

3.3 GASPI segments

GASPI does not enforce a specific memory model, like, for example, the symmetric distributed memory management of OpenSHMEM [9]. Rather GASPI offers PGAS in the form of configurable RDMA pinned memory segments. Modern hardware typically involves a hierarchy of memory with respect to the bandwidth and latencies of read and write accesses. Within that hierarchy are non-uniform memory access (NUMA) partitions, solid state devices (SSDs), graphical processing unit (GPU) memory or many integrated cores (MIC) memory. In Fig. 1

two GASPI segments were created, one segment across the memory of the Xeon Phi and one segment across the main memory of the x86 host. It also would have been possible to create the segment on Xeon Phi and e.g. a segment per NUMA domain of the x86 host. In general, the GASPI memory segments can serve as an abstraction which is able to represent any kind of memory level, mapping the variety of hardware layers to the software layer. A segment is a contiguous block of virtual memory. In the spirit of the PGAS approach, these GASPI segments may be globally accessible from every thread of every GASPI process and represent the partitions of the global address space. The segments can be accessed as global, common memory, whether local - by means of regular memory operations - or remote, by means of the communication routines of GASPI. Memory addresses within the global partitioned address space are specified by the triple consisting of the rank, segment identifier and the offset. Allocations inside of the pre-allocated segment memory are managed by the application.

3.4 GASPI queues

GASPI provides the concept of message queues. These queues facilitate higher scalability and can be used as channels for different types of requests where similar types of requests are queued and then get synchronized together but independently from the other ones (separation of concerns, e. g. one queue for operations on data and another queue for operations on meta data). The several message queues guarantee fair communication, i. e. no queue should see its communication requests delayed indefinitely.

3.5 GASPI one sided communication

One sided communication in GASPI is handled through asynchronous RDMA calls in the form of RDMA read and RDMA write. GASPI provides extended functionality for these calls which allows arbitrarily strided data with arbitrary offsets and length to be written - even with an equally arbitrarily strided (user-defined) transposition on the remote side if required.

3.6 GASPI weak synchronization

One-sided communication procedures have the characteristics that the entire communication is managed by the local process only. The remote process is not involved. This has the advantage that there is no inherent synchronization between the local and the remote process in every communication request. Nevertheless, at some point, the remote process needs the information as to whether the data which has been sent to that process has arrived and is valid. To this end, GASPI provides so-called weak synchronization primitives which update a notification on the remote side. The notification semantic is complemented with routines which wait for an update of a single or even an entire set of notifications. In order to manage these notifications in a thread-safe manner GASPI provides a thread safe atomic function to reset local notification with a given ID. The atomic function returns the value of the notification before reset. The notification procedures are one- sided and only involve the local process.

3.7 GASPI passive communication

GASPI provides a single routine with two-sided semantics in the form of passive communication. Passive communication aims at communication patterns where the sender is unknown (i.e. it can be any process from the receiver perspective) but there is a potentially need for synchronization between processes. Passive communication typically happens completely asynchronously to the

rest of the application workflow and is most directly comparable to a non-time critical active message. Example use cases are distributed updates or the logging of results to the console. The passive keyword means that the communication calls avoid busy-waiting, computation. Instead passive receives are triggered by an incoming message in a predefined communication queue.

3.8 GASPI global atomics

GASPI provides atomic counters, i. e. globally accessible integral types that can be manipulated through atomic procedures. These atomic procedures are guaranteed to execute from start to end without fear of preemption causing corruption. GASPI provides two basic operations on atomic counters: `fetch_and_add` and `compare_and_swap`. Global atomics can be applied to all data in the GASPI segments.

3.9 GASPI collective communication

Collective operations are operations which involve a whole set of GASPI processes. Collective operations can be either synchronous or asynchronous. Synchronous implies that progress is achieved only as long as the application is inside of the call. The call itself, however, may be interrupted by a timeout. The operation is then continued in the next call of the procedure. This implies that a collective operation may involve several procedure calls until completion. We note that collective operations can internally also be handled asynchronously, i.e. with progress being achieved outside of the call. Beside barriers and reductions with predefined operations, reductions with user defined operations are also supported via callback functions.

4 Conclusion

We have presented the Global Address Space Programming Interface (GASPI) as an alternative to the MPI programming model. GASPI is a partitioned global address space API, targeting both extreme scalability and failure-tolerance. With GPI 2.0, the GASPI standard has been implemented by Fraunhofer ITWM as a highly efficient open source implementation under GPL v3. We cannot give a full account of the performance specifications of GPI 2.0, here. These may be inspected on the GPI site of Fraunhofer ITWM [3]. We note however, that many of these results are close to the wire speed. We also would like to refer to the GASPI website [4].

5 Acknowledgment

The authors would like to thank the German Ministry of Education and Science for funding the GASPI project (funding code 01IH11007A) within the program ICT 2020 - research for innovation. Further more, the authors are grateful to all of their project partners for having fruitful and constructive discussions.

References

[1] Bradford L. Chamberlain, David Callahan, Hans P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007.

[2] UPC Consortium. UPC Specifications v1.2. *Lawrence Berkeley National Lab Tech Report LBNL-59208*, 2005.

[3] Fraunhofer ITWM. GPI - Global Address space programming Interface. http://www.gpi-site.com.

[4] GASPI Consortium. GASPI: Specification of a PGAS API. http://www.gaspi.de.

[5] MPI Forum . MPI: A MESSAGE PASSING INTERFACE STANDARD.

[6] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su and K. Yelick. Titanium Language Reference Manual. *U.C. Berkeley Tech Report UCB/EECS-2005-15*, 2005.

[7] R. Machado and C. Lojewski. The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. *Computer Science - Research and Development*, 23:125–132, 2009. 10.1007/s00450-009-0088-2.

[8] R.W. Numrich and John Reid. Co-array Fortran for parallel programming. *ACM Fortran Forum*, 1998.

[9] Stephen W. Poole, Oscar Hernandez, Jeffery A. Kuehn, Galen M. Shipman, Anthony Curtis, and Karl Feind. OpenSHMEM - Toward a Unified RMA Model. *Encyclopedia of Parallel Computing*, pages 1379–1391, 2011.

Improving Communication Through Loop Scheduling in UPC

Michail Alvanos[1,2,3], Gabriel Tanase[4], Montse Farreras[2,1], Ettore Tiotto[5], José Nelson Amaral[6] and Xavier Martorell[2,1]

[1] Programming Models, Barcelona Supercomputing Center
malvanos@bsc.es
[2] Department of Computer Architecture Universitat Politècnica de Catalunya
{mfarrera,xavim}@ac.upc.edu
[3] Centre for Advanced Studies Research, IBM Canada
[4] IBM Research Division, Thomas J. Watson Research Center
igtanase@us.ibm.com
[5] Static Compilation Technology, IBM Toronto Laboratory
etiotto@ca.ibm.com
[6] Department of Computing Science, University of Alberta
jamaral@ualberta.ca

Abstract

Partitioned Global Address Space (PGAS) languages appeared to address programmer productivity in large scale parallel machines. The main goal of a PGAS language is to provide the ease of use of shared memory programming model with the performance of MPI. Unified Parallel C programs can suffer from shared access conflicts on the same node. Manual or compiler code optimization is required to avoid oversubscription of the nodes. This paper explores loop scheduling algorithms and presents a compiler optimization that schedules the loop iterations to provide better network utilization and avoid node oversubscription.

1 Introduction

Unified Parallel C language [7] promises simple means for developing applications that can run on parallel systems without sacrificing performance. One of the inherited limitations of the UPC languages is the transparency provided to the programmer when accessing shared memory. Furthermore, high-radix network topologies are becoming a common approach [4, 5] to address the latency wall of modern supercomputers. To effectively avoid network congestion the programmer or the runtime spreads the non-uniform traffic evenly over the different links. This paper explores possible loop scheduling schemes and proposes a loop transformation to improve the performance of the network communication. The compiler [6] applies a loop transformation to spread the communication between different networks and to avoid overloading computational nodes.

The researchers at Universitat Politècnica de Catalunya and Barcelona Supercomputing Center are supported by the IBM Centers for Advanced Studies Fellowship (CAS2012-069), the Spanish Ministry of Science and Innovation (TIN2007-60625, TIN2012-34557, and CSD2007-00050), the European Commission in the context of the HiPEAC3 Network of Excellence (FP7/ICT 287759), and the Generalitat de Catalunya (2009-SGR-980). IBM researchers are supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. The researchers at University of Alberta are supported by the NSERC Collaborative Research and Development (CRD) program of Canada.

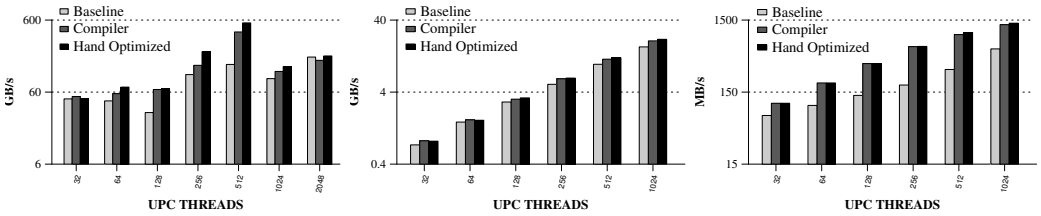


Figure 1: Comparison of compiler-transformed and hand-optimized code: upc_memput (a), fine-grained get (b), and fine-grained put (c).

2 Compiler-assisted loop transformation

The idea of compiler-assisted loop transformation is to hide the network complexity from the programmer's perspective. First, the compiler collects normalized loops that have no loop carried dependencies and that contain shared references. Next, the compiler checks if the upper bound of the loop is greater or equal the number of UPC threads, and if it is not `upc_forall` loop. The compiler categorizes the loops in two categories based on the loop upper bound and shared access type:

- Loops that have coarse-grain transfers and whose upper bound is the number of UPC threads. In this case, the compiler creates a lookup table with the number of UPC threads as the array size. The contents of the lookup table are the randomly shuffled threads. Then the compiler replaces the induction variable inside the body of the loop with the return value of the look up table.
- Loops that contain fine-grained communication or contain coarse-grain transfers but with the upper bound of the loop different from the number of threads. The compiler skews the iterations in such a way that each UPC thread starts executing from a different point in the shared array.

3 Experimental Results

The evaluation uses one microbenchmark and four applications in a IBM Power 775 [5]. The microbenchmark is a loop that accesses a shared array of structures. There are three variations of this microbenchmark. In the **upc_memput** microbenchmark the loop contains coarse grain `upc_memput` calls. The **fine-grained get** contains shared reads and the **fine-grained put** contains shared writes.

While the performance of the manual and compiler-transformed fine-grained microbenchmarks is similar, the compiler transformation achieves slightly lower performance than the hand-optimized benchmark because of the insertion of runtime calls. Figure 1 compares the compiler-transformed and hand-optimized code.

Cache level	Baseline %	Schedule
Level 1	0.14%	0.19%
Level 2	0.19%	24.49%
Level 3	0.32%	28.84%

Table 1: Average cache misses using 256 UPC threads for Sobel.

There are three different patterns in the applications. In the first category are applications that have performance gain compared with the version without

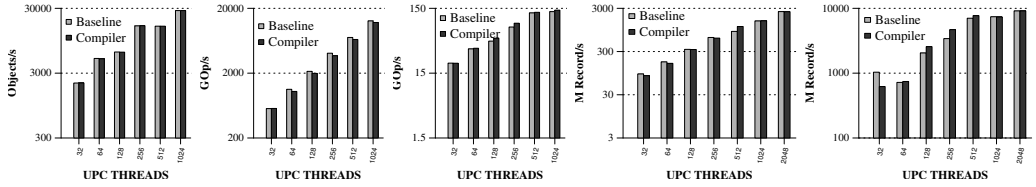


Figure 2: Comparison of baseline and compiler-transformed code for fish (a), Sobel (b), NAS FT (c), bucket-sort (d) and bucket-sort with only the communication pattern (e).

the scheduling (baseline), such as the NAS FT benchmark [3]. This benchmark achieves from 3% up to 15% performance gain, due to its all-to-all communication pattern. Moreover, the performance of the gravitational gravitational Fish benchmark [1] is almost identical and the transformation reveals minimal performance gains. On the other hand, the performance of the Sobel benchmark [3] decreases up to 20% compared with the baseline version, because of poor cache locality. Table 1 presents the cache misses for the Sobel benchmark for different cache levels using the hardware counters. Figures 2(d) and 2(e) present the results for bucket-sort [2] with enabled and disabled local sort. There are minor differences between the baseline and the transformed version when using the benchmarks with enabled the local sort. However, the transformed version has better results — up to 25% performance gain — than the baseline version when only the communication part is used.

4 Conclusions and Future Work

This paper presents an optimization to increase the performance of different communication patterns using loop iterations scheduling. The compiler optimization improves the performance of programs that contain problematic access patterns, including the all-to-all communication.

References

[1] S.J. Aarseth. *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge Monographs on Mathematical Physics. Cambridge University Press, Cambridge, U.K.; New York, U.S.A., 2003.

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2001. ISBN 0262032937.

[3] Tarek El-Ghazawi and Francois Cantonnet. UPC performance and potential: a NPB experimental study. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–26.

[4] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, James Reinhard, et al. Cray cascade: a scalable HPC system based on a Dragonfly network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 103. IEEE Computer Society Press, 2012.

[5] R. Rajamony, LB Arimilli, and K. Gildea. PERCS: The IBM POWER7-IH high-performance computing system. *IBM Journal of Research and Development*, 55(3):3–1, 2011.

[6] Gabriel Tanase, Gheorghe Almási, Ettore Tiotto, Michail Alvanos, Anny Ly, and Barnaby Daltonn. Performance Analysis of the IBM XL UPC on the PERCS Architecture. Technical report, 2013. RC25360.

[7] UPC Consortium. UPC Specifications, v1.2. Technical report, Lawrence Berkeley National Lab LBNL-59208.

Posters

These short descriptions give a synopsis of the works that were accepted for a poster presentation.

The performance of the PCJ library for massively parallel computing

Marek Nowicki[1], Piotr Bała[1,2]

[1] Nicolaus Copernicus University, Toruń, Poland
[2] ICM, University of Warsaw, Warsaw, Poland
{faramir@mat.umk.pl, bala@icm.edu.pl

This paper presents a new version of the PCJ library [1, 2] for Java language that helps to perform parallel and distributed calculations. The current version is able to work on the multicore systems connected with the typical interconnect such as ethernet or infiniband providing users with the uniform view across nodes.

The library implements partitioned global address space model and was inspired by languages like Co-Array Fortran, Unified Parallel C and Titanium. In contrast to listed languages, the PCJ does not extend nor modify language syntax. When developing the PCJ library, we put emphasis on compliance with Java standards. The programmer does not have to use additional libraries, which are not part of the standard Java distribution.

In the PCJ, each task has its own local memory and stores and access variables only locally. Some variables can be shared between tasks and that variables can be accessed, read and modified by other tasks. The library provides methods to perform basic operations like synchronization of tasks, get and put values in asynchronous one-sided way. Additionally the library offers methods for creating groups of tasks, broadcasting and monitoring variables.

We have used PCJ library to paralelize example application, in this case ray tracing. We have measured the performance of 3D ray tracing of the scene rendered at a resolution of NxN pixels. The PCJ version of the *RayTracer* test contains a naive implementation of *Reduce* operation that uses asynchronous get (*getAsync*) method. The speedup for PCJ library is competitive compare to MPICH2; the gained speedup for larger cases is even better.

The tests show that there are still areas for improvements, especially in the case of internode communication. The mechanisms for synchronizing and transmitting messages should be improved. Additionally, there are no advanced techniques for the breakdown recovery and node failure handling.

References

[1] PCJ library is available from the authors upon request.

[2] M. Nowicki, P. Bała. Parallel computations in Java with PCJ library In: W. W. Smari and V. Zeljkovic (Eds.) *2012 International Conference on High Performance Computing and Simulation (HPCS)*, IEEE 2012 pp. 381-387

A Comparison of PGAS Languages on Multi-core Clusters

Konstantina Panagiotopoulou and Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh EH144AS, UK

The class of Partitioned Global Address Space (PGAS) languages aims to deliver an increased level of abstraction to classical supercomputing languages, while retaining their high performance. However, with abstraction comes additional run- and compile-time overhead, and a reduced portfolio of techniques to tune the parallel performance. In this work we study this tension on three state-of-the-art PGAS languages, X10, Chapel, and UPC, and we report on programmability, tunability and parallel performance, for a typical, data-parallel application.

This work, which is part of an MSc final project, aims to *evaluate the degree in which PGAS languages achieve the goals of portability, programmability, performance and robustness*, using X10, Chapel, and UPC as our main languages of interest. To this end, we present cluster-based implementations of the N-body problem in all three languages, and we give an assessment of performance, language features and tool support.

Preliminary Results: For our N-body implementations we achieve the best speedup of 5.4 on 12 machines using X10. Our hardware platform is a 12-node cluster of 8-core machines (2 quad-core Xeon E5506 2.13GHz, with 256kB L2 and 4MB shared L3 cache). The implementations in the other languages are still in a performance debugging phase, and so-far both UPC and Chapel deliver speedups of ca. 2 on multiple machines. Details will be on the poster.

X10 has a strong object-oriented flavour, which helps a programmer, not already familiar with the details of parallel programming, in developing an initial version. The syntax is slightly unusual, combining object oriented syntax conventions (e.g. array declaration) with higher-order functions (e.g. array initialisation), as used in functional programming. In exploring the coordination aspects of the execution, we have reached the point were low-level constructs had to be resorted to in order to reduce the communication overhead. This was unexpected, but potentially due to the limited time spent on this implementation of the code.

The tutorial and community support for Chapel proved much more useful than those for the other languages, but to some degree they are also confusing, due to different language versions. The main web page helps in that respect, highlighting the most important information. However, details of example applications are not elaborated on, and tutorials sometimes admit that low-level issues, such as data distribution, have been elided in the tutorial. While it is understandable that the community focuses on the selling points of the language, it is this part of "awkward" low-level issues that take a substantial amount of programmer time for a new programmer to get started in this field.

From a language design point of view, Chapel's concepts of (sub-)domains proved very useful for the N-body implementation, avoiding manual decomposition and communication. Defining the array distribution required experimentation, due to the many options available to e.g. block distributions, and changes in syntax. Higher-order functions proved to be an important structuring element in the program design.

As expected, the UPC version is the most verbose of the parallel implementations. It is similar to the Chapel implementation, using a parallel for loop, but lacks Chapel's support for coordination, domains or bulk operations on arrays.

Communication and Computation Overlapping Parallel Fast Fourier Transform in Cray UPC

Erik Järleberg[1], Johannes Bulin[1], Jens Doleschal[2], Stefano Markidis[1] *and Erwin Laure[1]

[1] HPCViz Department, KTH Royal Institute of Technology, Stockholm, Sweden
erikjar@kth.se, bulin@kth.se, markidis@kth.se, erwinl@kth.se
[2] Center for Information Services and High Performance Computing, Technische Universität Dresden, Dresden, Germany
jens.doleschal@tu-dresden.de

The numerical computation of the Fourier Transform, or the Discrete Fourier Transform (DFT), is an important part in a myriad of different scientific contexts. Applications range from time series and spectral analysis, to solution of partial differential equations, signal processing and image filtering. From the original method devised in 1965 by Cooley and Tukey, several different algorithms for calculating the 2D/3D Fast Fourier Transform (FFT) have been developed through the years. Fundamentally, two basic parallel formulations exist: the binary exchange algorithm and the transpose algorithm. In this poster, we use the second formulation, i.e where a transpose operation dominates the communication part of the algorithm. The NAS benchmark is a suite of different benchmark kernels exhibiting different communication patterns. In this poster, we work on the Fourier Transform (FT) kernel. UPC (Universal Parallel C) is a Partitioned Global Address Space (PGAS) language, which means that all processes have access to the same global address space, so that a process can read and write data to the memory of another remote process. The one-sided nature of communication in UPC, provides the language with a certain edge over traditional message-passing-based paradigms such as MPI: the overhead for small messages sizes is significantly reduced in UPC if compared to MPI. Due to the large overhead costs for smaller sizes packages in MPI, one normally tends to aggregate several messages into larger bulk messages, which can then be more efficiently transferred over the network. This is not necessary to the same extent in UPC, and this allows one to optimize algorithms in novel ways, by increasing the total number of messages sent, yet still retaining good performance.

In this work, we leveraged this feature of UPC to implement a communication-computation overlapped version of the NAS FT Benchmark, which allows us to achieve strong performance gains over the blocking version of NAS FT. These gains are solely possible due to the nature of one-sided communication in UPC. We modified the traditional UPC version of the NAS FT, and changed it from one bulk all-to-all transpose operation, into a non-blocking computation-communication overlapped version. The original NAS FT UPC version with a D-class input size, using 128 cores, runs in 268 seconds, which can be contrasted with the modified non-blocking version of the same size, also using 128 cores, which runs in 185 seconds, which is roughly 30% faster. We found that the combined use of an algorithm that allows to overlap communication and computation and the use of non-blocking DMAPP remote memory access lead to a decrease of the execution of time if compare to a blocking version of the FFT. We report that the non-blocking FFT is 30% faster than blocking FFT on the 128 cores, while on 1024 cores the non-blocking version is 2% faster.

*acknowledges support from the European Commissions Seventh Framework Programme (FP7/2007-2013) under the grant agreements no. 287703 (CRESTA, cresta-project.eu).

Validation and Verification Suite for OpenSHMEM

Swaroop Pophale[1], Tony Curtis[1], Barbara Chapman[1] and Stephen Poole[2]

[1] University of Houston Houston, Texas, USA
spophale,tonyc,chapman@cs.uh.edu
[2] Oak Ridge National Lab Oak Ridge, Tennessee, USA
spoole@ornl.gov

Abstract

With the evolution of many core and multicore systems different programming models have evolved to meet the needs of the current applications and their communication and computation needs. One such programming model is the Partitioned Global Address Space model which works to combine the advantages of both shared and distributed memory programming by providing a finer control over the data placement and access. OpenSHMEM is a PGAS library that allows for programmers to develop parallel SPMD applications with the help of its point-to-point (putget) communication operations, remote atomic memory operations, collective operations (like broadcast and reduce) and a simple set of ordering, locking, and synchronization primitives. The OpenSHMEM Specification provides general guidelines and expectation of the library's behavior but without a validation suite the different vendor implementations could differ in their interpretation of the specification. In this paper we present the first Validation and Verification suite which not only validates an OpenSHMEM library's functions but also provides micro-benchmarks that can be used to study and compare the performance of each of the individual library APIs for further analyses across different OpenSHMEM library implementations or different hardware configurations.

1 Introduction

In parallel computing over many and multi-core systems the Partitioned Global Address Space programming model has been largely successful in delivering optimum performance for distributed systems running SPMD applications. Popular language extensions and libraries under the PGAS programming model include Coarray Fortran (CAF), Unified Parallel C (UPC), OpenSHMEM etc. While CAF and UPC are language extensions, OpenSHMEM [2] is a library which provides a library API for one-sided point-to-point communication operations, remote atomic memory operations, collective operations (like broadcast and reduce) and a simple set of ordering, locking, and synchronization primitives. Till the OpenSHMEM specification V 1.0 [1] was finalized in 2011, the libraries were historically called SHMEM and had many variant flavors depending on the vendor and the targeted hardware system. These SHMEM library implementations vary in optimizations and hardware support they provide, but they all uphold the key concepts and principles of the original SHMEM library developed in 1993 by Cray Research Inc. Since the announcement of the specification, all SHMEM library implementations are striving to come together to support a unified API that will promote portability of applications and reproducibility of results across different (now) OpenSHMEM libraries that may run on different hardware platforms. To aid this effort University of Houston, along with Oak Ridge National Lab have developed the Validation and Verification (V&V) suite that checks for conformance of a given OpenSHMEM library to the current OpenSHMEM specification. In this poster we give an introduction to the OpenSHMEM specification and the key concepts of

OpenSHMEM which are the foundation of the Validation and Verification suite. We discuss the different OpenSHMEM library calls and the expected behavior of an implementation. We also deliberate over the different ways in which we verify that an OpenSHMEM library adheres to the specification and the challenges that we faced therein.

References

[1] Openshmem specification 1.0 (openshmem.org).

[2] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.